Ashling Product Brief APB217
v1.0.22, 30th September 2015

Using Vitra-XD for ARM with Mentor Graphics Sourcery CodeBench

Contents

1. Introduction

This Ashling Product Brief (APB217)describes the usage of Ashling's Vitra-XD with Mentor Graphics Sourcery CodeBenchIDE. Vitra-XD is a powerful high-speed, high-capacity trace (500+ GB) and run-time control debug probe for embedded development on ARM™ RISC cores. Vitra-XD works with Mentor Graphics Sourcery CodeBench aneclipse based IDE and allows:

- Capturing of program-flow and data-accesses in real-time, non-intrusively
- Exercising the program in target (go, step, halt, breakpoints, interrogate memory, registers and variables)

Figure 1.Vitra-XD

Vitra-XD supports Gigabit Ethernet as well as USB2.0 High-speed host connections. This APB will look at using Vitra-XD and Sourcery CodeBench with a TI OMAP4460 based PandaBoard as shown below:

Figure 2. TI OMAP4460 PandaBoard

In addition, Section 7 provides an overview of using Vitra-XD with the Xilinx ZC702 board.

2. Installation and Configuration

2.1 Installing Ashling components

If you are installing Sourcery CodeBench for the first time, ensure you install the required Vitra-XD plug-ins by selecting the **Ashling Vitra-XD Trace** components in the Sourcery CodeBench installer as shown below. Note that if you have an existing Sourcery CodeBench installation, you should uninstall and reinstall with the Ashling Vitra-XD Trace plug-ins.

Please note: administrator rights are required to install ALL of the Ashling components.

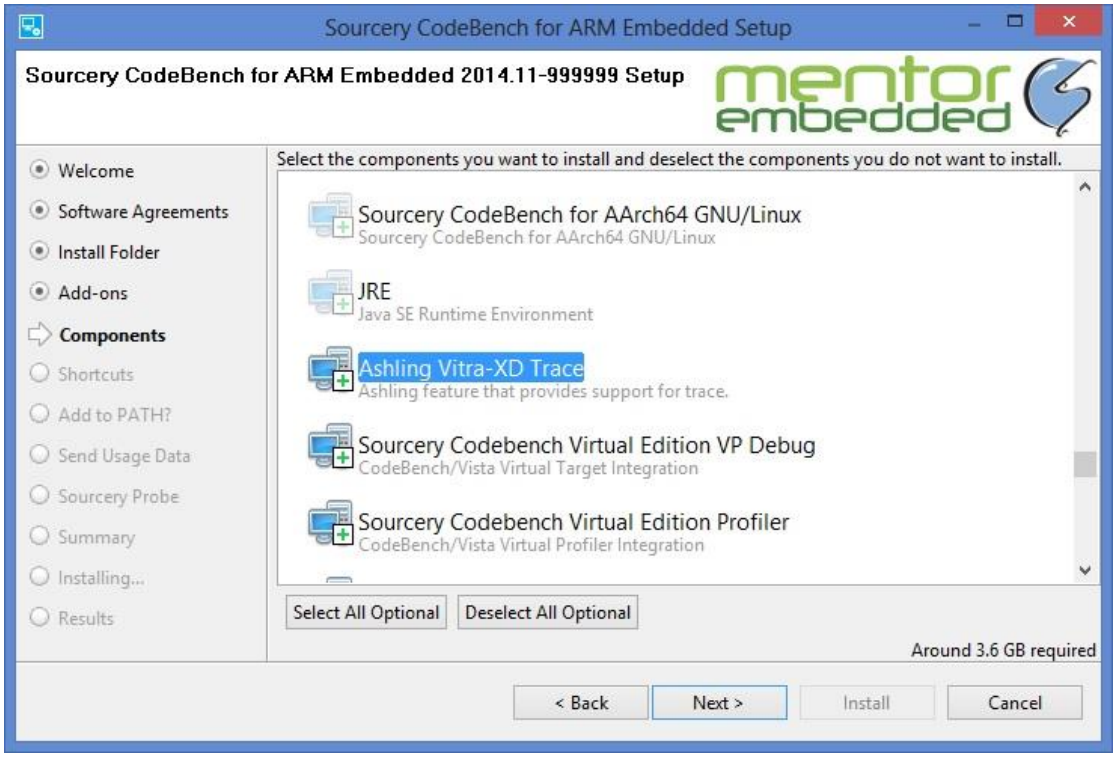


Figure 3. Selecting Ashling Vitra-XD Trace components

Note that if installation is done without selecting this option then a complete reinstallation of Sourcery CodeBench is required to install the Ashling Vitra-XD Trace components.

2.2 Quick-start summary

Using the Ashling Vitra-XD with Sourcery CodeBench requires:

1. Installation of MariaDB SQL server database which is used for storing captured Vitra-XD Trace data on the host PC. See section**2.3**

2. Depending on how Vitra-XD is connected to your PC, then you will need to either:
   a. Install Vitra-XD USB driver. See section **2.4** .
   b. Configure Vitra-XD ethernet. See section**2.5** .

3. Once the above are complete you are ready to start debugging (see section3) and tracing (see section**4**)with Sourcery CodeBench and Vitra-XD.

2.3 Installing the MariaDB SQL Server Database

Vitra-XD uses the MariaDB 5.5 SQL database server for storing trace data. This must be manually installed on your PC. Separate instructions are provided for Windows (see section **2.3.1**) and Linux hosts (see section **2.3.2**). Once installed, you need to configure Sourcery CodeBench to use the installed SQL Server (see section **2.3.3**)

2.3.1 Installing MariaDB Server on Windows Hosts

Execute the SQL Components installer(*with administrator rights*) located at Sourcery CodeBench installation folder e.g. `<Installation_ path>\codebench\libexec\arm-none-linux-gnueabi-post-install\sprite-drivers`. After installation, ensure that the MariaDB service is running by executing the command `services.msc` via the Windows **Run** menu and confirming that the `AshMariaDB` service has started.

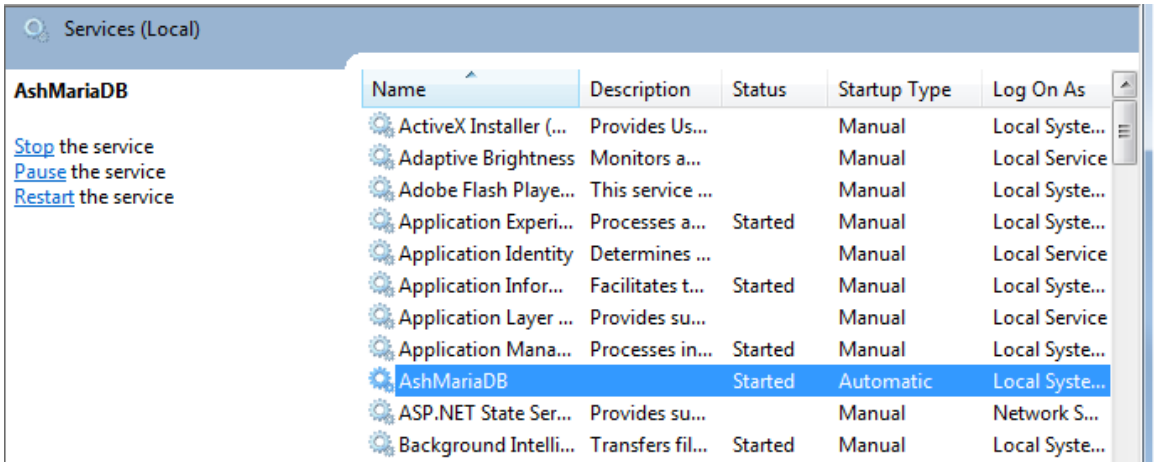


Figure 4. AshMariaDB service

2.3.2 Installing MariaDB Server on Linux Hosts

There are two steps required for installing MariaDB Server:

1. Installing MariaDB
2. Installing the ODBC drivers

2.3.2.1 Installing MariaDB (Linux Hosts)

To install the MariaDB server, follow the instructions provided at:

https://downloads.mariadb.org/mariadb/5.5.30/

Note: The MariaDB Server installer prompts you to set a password for the MariaDB `root` user. The Sourcery CodeBench IDE defaults to "`vitraxd`". If you wish to set a different password, then it needs to be changed in Sourcery CodeBench IDE preferences (**Window|Preferences|Vitra-XD Setup)** section also. After installation, update the file `/etc/mysql/my.cnf` *as follows:*

```
[mysqld]
#
# * Basic Settings
#
user            = mysql
pid-file        = /var/run/mysqld/mysqld.pid
socket          = /var/run/mysqld/mysqld.sock
port            = 3306
basedir         = /usr
datadir         = /var/lib/mysql
tmpdir          = /tmp
default_storage_engine = myisam
lc_messages_dir = /usr/share/mysql
```

Figure 5. Sample my.cnf after modification

Ensure that there are no other `default_storage_engine` statements in the file (comment them out by preceding them with a '`#`' if necessary). Now start the MariaDB server as follows:

```
$sudo service mysql start
```

2.3.2.2 Installing the ODBC drivers(Linux Hosts)

To install the ODBC driver, download the source packages from http://www.unixodbc.org/ and follow the instructions specified in the *Download* page. You may download and install the MySQL ODBC connector from http://dev.mysql.com/downloads/connector/odbc/.

Install the ODBC connector as follows:

```
$ myodbc-installer -d -a -n "MySQL ODBC 5.3Driver" -t "DRIVER=/usr/lib/libmyodbc5w.so"

$ myodbc-installer -s -a -n "AshlingDB" -t "DRIVER=MySQL ODBC 5.3Driver;SERVER=localhost;Socket=/var/run/mysqld/mysqld.sock"
```

Note:

1. For 64-bit Linux host systems, you need to have 32-bit versions of unixODBC and MySQL ODBC connector.
2. Ensure that the environment variables `ODBCINI` and `ODBCSYSINI` are pointing to the right `odbc.ini` file. For example, if your `odbc.ini` file is located in `/usr/local/etc`, set the environment variables like this:
   ```
   export ODBCINI=/usr/local/etc/odbc.ini
   export ODBCSYSINI=/usr/local/etc
   ```

2.3.3 Configuring Sourcery CodeBench to use the Ashling SQL Components

Open the **Window|Preferences** menu item and select the **Vitra-XD Setup** section:

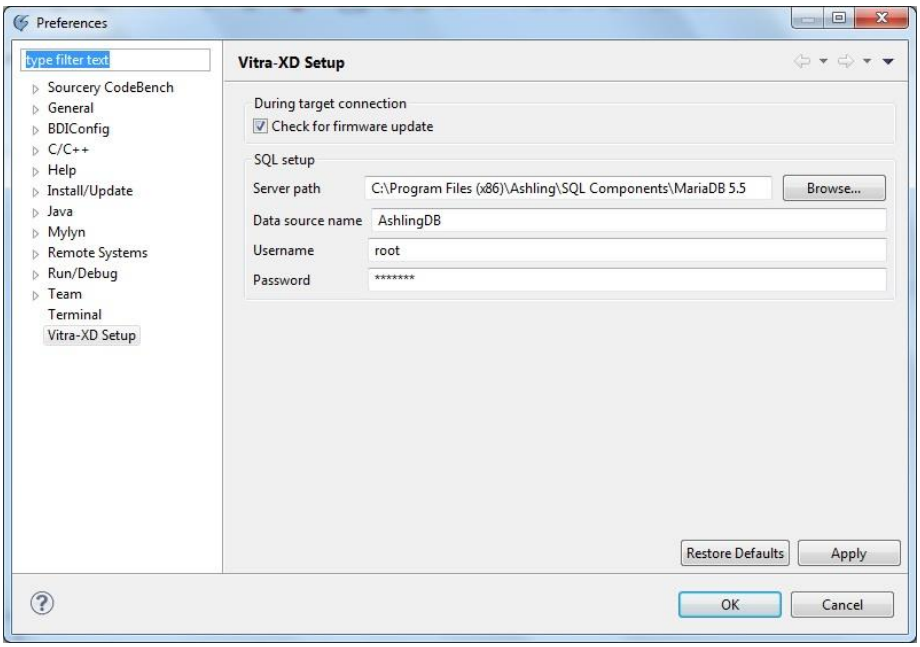


Figure 6. Configure Ashling SQL settings

Adjust the **SQL setup** section based on the settings used during the installation.

Server path	For Windows hosts this is the full path to where the Ashling SQL Components were installed e.g. `C:\Program Files\Ashling\SQL Components\MariaDB 5.5\`or `C:\Program Files (x86)\Ashling\SQL Components\MariaDB 5.5\` for 64-bit Windows. For Linux hosts this is the path to the mysql directory created by the MariaDB installation e.g. `/etc/mysql/`
Data source name	Name of the SQL database to use. Use default.
Username	Username for accessing the database. Use default.
Password	Password for accessing the database. Use default (`vitraxd`).

2.4 Using USB with Vitra-XD

2.4.1 Windows™ USB Driver Installation

When you first connect Vitra-XD to your PC, you will get a **New USB hardware found** message and will be prompted to install the appropriate USB drivers. The Ashling Vitra-XD drivers are installed in your installation directory. e.g. *`<Installation_path>\codebench\i686-mingw32\arm-none-eabi\ash\drivers\usb`*. Direct the Windows **Hardware Installation Wizard** to this directory so that it can locate the necessary drivers and complete the installation. Windows only needs to perform this operation the first time you connect your Vitra-XD to the PC. The Vitra-XD USB driver is called `libusb0.sys` (`libusb0_x64.sys`for 64-bit operating systems).

2.4.2 Linux x86 USB Driver Installation

Vitra-XD uses the `libusb` driver (http://libusb.sourceforge.net/). By default, the driver is stored in: `/usr/lib`

Check for this as follows:

```
$ ls /usr/lib/libusb* /usr/include/usb.h
```

If you see `/usr/include/usb.h` and `libusb-0.1.so.4.4.0` or higher, then they are installed on your system and you can skip the next section on libusb installation

Please note: If/usr/lib/libusb directory does not include a file titled libusb.so (exact filename), then create a symlink as follows:

```
$ln -s /usr/lib/libusb-0.1.so.4.4.0 /usr/lib/libusb.so
```

2.4.2.1 Ubuntu/Debian libusb installation

Install libusb using the following command:

```
$ sudo apt-get install libusb-dev
```

Note: For 64-bit Linux host systems, the 32-bit version of libusb has to be installed using the following command:

```
$ sudo apt-get install libusb-dev:i386
```

If /usr/lib/libusb directory does not include a file titled libusb.so(exact filename), then create a symlink as follows:

```
$ln -s /usr/lib/libusb-0.1.so.4.4.0 /usr/lib/libusb.so
```

2.4.2.2 Fedora/other distribution libusb installations

Download the latest libusbfromhttp://libusb.sourceforge.net/ and install as follows:

```
$ tar zxf libusb-0.1.12.tar.gz (use appropriate version number)
$ ./configure --prefix=/usr
$ make
$ make install
```

If /usr/lib/libusb directory does not include a file titled libusb.so(exact filename), then create a symlink as follows:

```
$ln -s /usr/lib/libusb-0.1.so.4.4.0 /usr/lib/libusb.so
```

2.4.2.3 Setting permissions

1. Ensure that Vitra-XD is connected to the PC, connected to the target and that the target is powered.
2. To ensure the current $USER has access to the Vitra-XD device, we recommend using the Linux utility udev (requires kernel 2.6 or later).
3. Ensure udev is installed and running on your system by checking for the udev daemon process (udevd) e.g.:
   ```
   $ ps –aef | grep udev
   ```
4. Create an udev rules file to uniquely identify the Vitra-XD device and set permissions as required by owner/ groups. An example udevfile is supplied (60-ashling.rules) which identifies Vitra-XD device (by Ashling's USB Product ID and Vendor ID). This file is located in your installation directory e.g.*<Installation_path>\codebench\i686-mingw32\arm-none-linux-gnueabi\ash\drivers\usb*.
5. The rules file must then be copied into the rules directory (requires root permission) e.g.:
   ```
   $ sudo cp ./60-ashling.rules /etc/udev/rules.d
   ```

2.5 Using Ethernet with Vitra-XD

By default, Vitra-XD powers up in DHCP mode and will acquire an IP address automatically if connected to a DHCP network. If a DHCP network is not available, then Vitra-XD will default to a Link-local address of 169.254.1.1/16. You may configure the IP address of a Vitra-XD using Sourcery CodeBench using the **Run|AshlingVitra-XD|Check for Vitra-XD(s)** menu item. This opens the following dialog box:

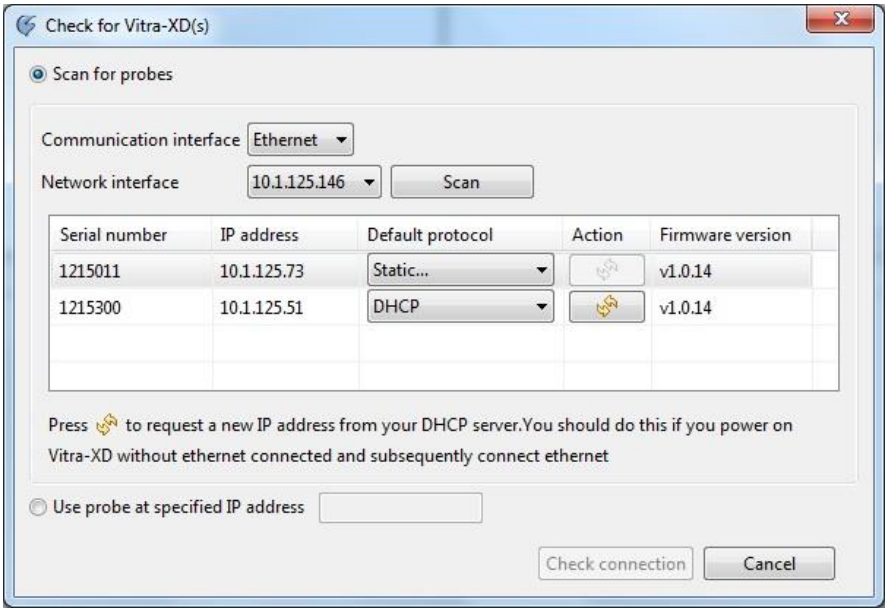


Figure 7. Configuring IP address of a Vitra-XD

Select the communication interface as **Ethernet**. If your PC/laptop has multiple network adaptors, select the appropriate one via **Network interface** and click the **Scan** button. This will initiate a search for all connected Vitra-XD's. Sourcery CodeBench internally scans using the multicast protocol (address: 226.0.01, receive-port 28007 and transmit-port 28008) to find the Vitra-XD probes connected to the network. If the probe is not listed, then you must manually enter the IP address of the probe you want to use. A Vitra-XD probe may not be discoverable due to:

- the probe is not on the same subnet as your current host machine
- your network adaptor is not configured for multi-cast
- the routers servicing your network/firewall are blocking multi-cast packets

2.5.1 Configuring Static IP for your Vitra-XD

You can configure a static IP address for your Vitra-XD irrespective of whether it is using USB or Ethernet as the current communication interface. Once you have done a scan and you have the probes listed, you can do this by clicking the **Default protocol** control as shown below.

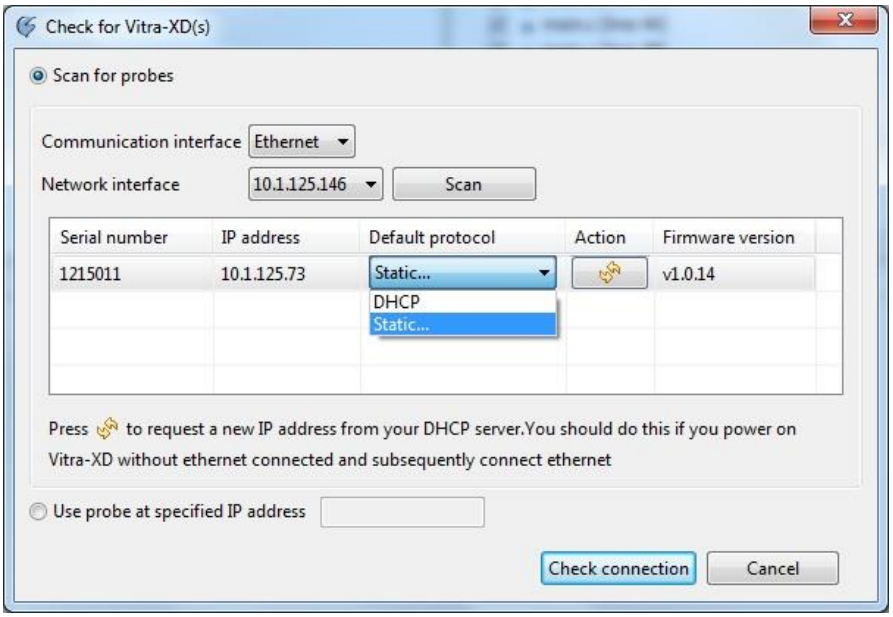


Figure 8. Configure Static IP address

If you select **Static**, a dialog box pops up as shown below, displaying the **IP address** and **Gateway**.

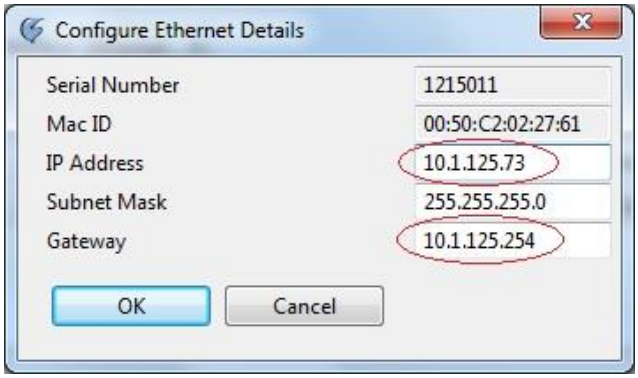


Figure 9. Setting Static IP details

Change the **IP address** and **Gateway** according to your settings and click **OK**.

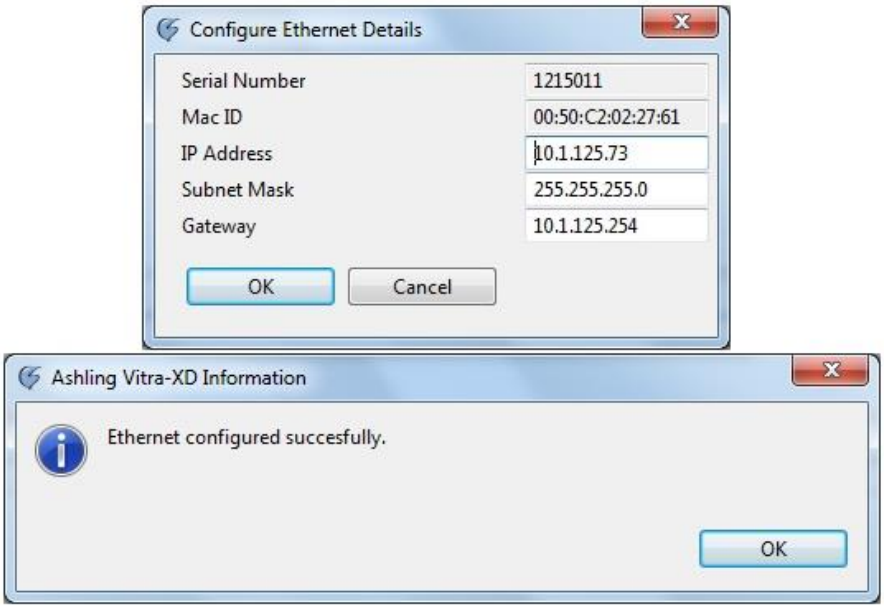


Figure 10.Changing Ethernet settings

2.6 Using Vitra-XD with Sourcery CodeBench on a x86 64-bit Linux host

To use Vitra-XD with Sourcery CodeBench on ax86 64-bit Linux host system the32-bit `libxml-2` package needs to be installed. Use the following command in an Ubuntu/Debian distribution:

```
$ sudo apt-get install libxml2:i386
```

3. Debugging with Sourcery CodeBench and Vitra-XD

3.1 Connecting Vitra-XD to the Target

Vitra-XD is designed to connect to the OMAP4460 PandaBoard via the supplied Target Probe Assembly (TPA) and 38-pin MICTOR extender cable as shown below:

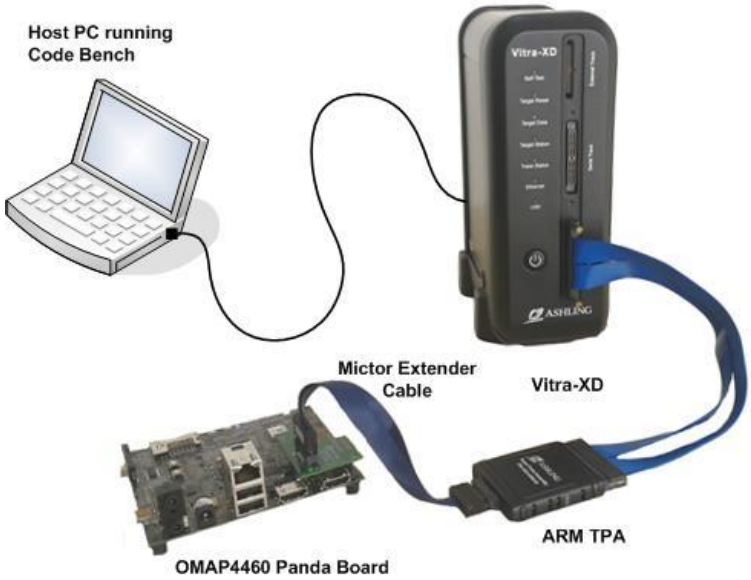


Figure 11. Vitra-XD connected to the TI OMAP4460 PandaBoard

The OMAP4460 PandaBoard does not have a 38-pin MICTOR connector; hence, you will need the AD-VXD-ARM38-PANDA adaptor from Ashling.

Please note the following recommended target connection sequence:

1. Ensure your target (OMAP4460 PandaBoard) is powered off.
2. Connect Vitra-XD to the target as shown above.
3. Power up Vitra-XD via the power button on the front-panel of Vitra-XD. The **Self Test** LED on Vitra-XD blinks orange during the self test process followed by green during the final initialisation process. When successfully completed, the **Self Test** LED is green.
4. Power up your PandaBoard and ensure its LED (D2)turns on (green)

3.2 Using Sourcery CodeBench

3.2.1 Getting started/configuring Sourcery CodeBench

1. To get started, run Sourcery CodeBench 

.

2. Sourcery CodeBench will then load. If this is your first-time running, then you will be prompted to specify your workspace (default directory for projects etc). Accept the default directory (or select an alternative).

3. A ready-to-use PandaBoard demo project is supplied e.g.
   <Installation_path>\codebench\i686-mingw32\arm-none-eabi\ash\demos\PandaDemo.zip.Follow the below steps to import the project into CodeBench.

4. Select **File|Import** and choose **General->Existing Projects into Workspace** option. Click **Next.**

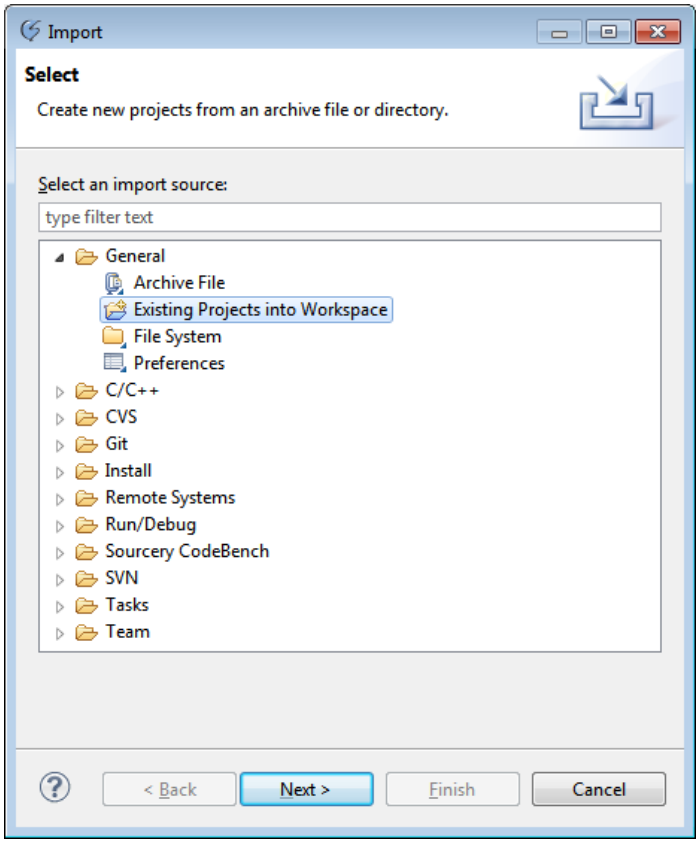


Figure 12. Importing the project

5. Choose **Select archive file** option and browse for the `PandaDemo.zip` file. Click **Finish**.

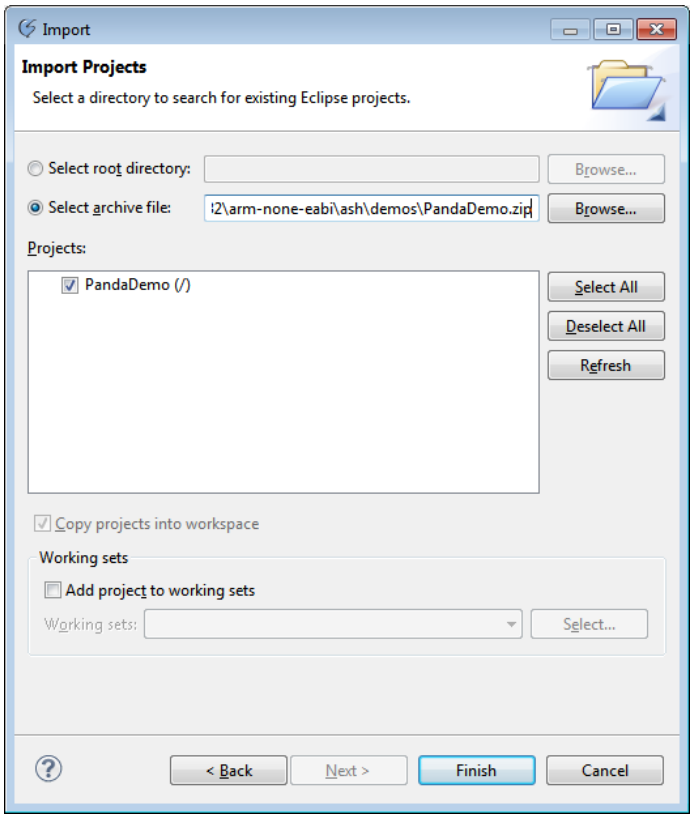


Figure 13. Select the archive file

6. The **Project Explorer** view shows the details of the project once it is imported.

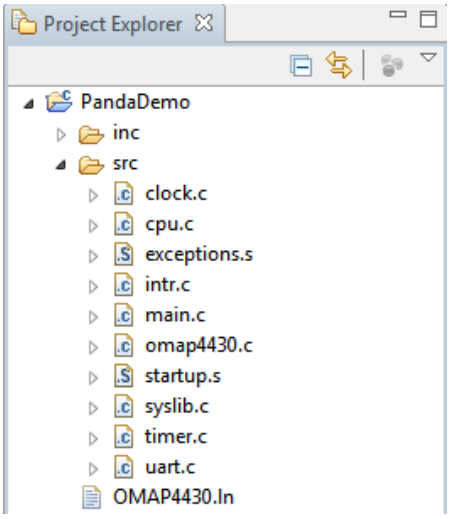


Figure 14. Project explorer view

7. Right click on the project and select **Build Project** option. This will build the project and create the executable. You can make changes in the source file and rebuild the application in similar way.

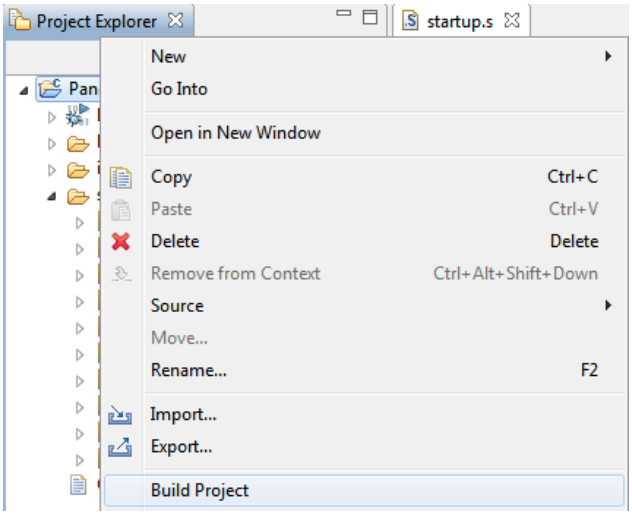


Figure 15. Build the project

8. After building the project, select **Run|Debug Configurations** and select the **Debugger** tab.

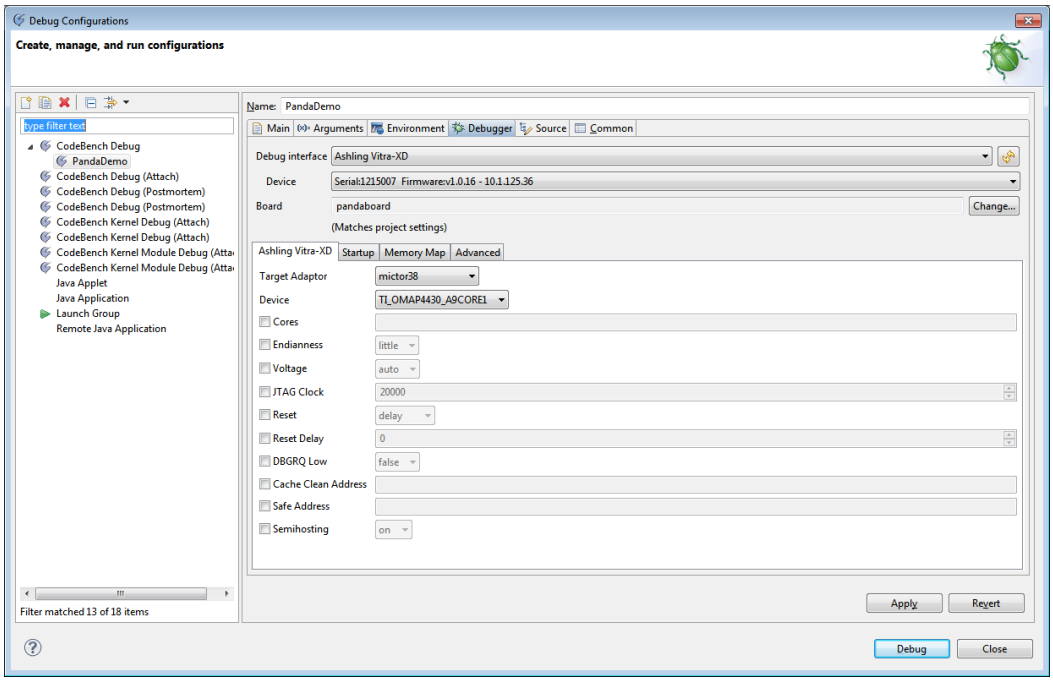


Figure 16. Debugger tab

Select **Ashling Vitra-XD** as **Debug interface** and select the required Vitra-XD probe from the auto-scanned list of probes based on **Serial** number.

Select the **Board** in which the application has to be downloaded and debugged.

9. Select **Ashling Vitra-XD** tab in the **Debugger Options** group.

10. Select **Device** as**TI_OMAP4430_A9CORE1**.(Defaults will work in the case of TI PandaBoard.)

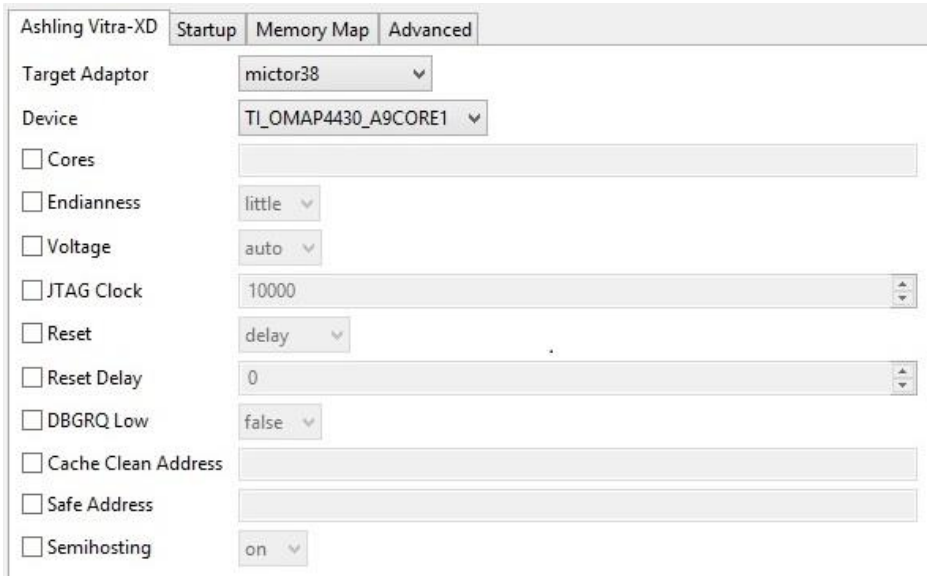


Figure 17. Debugger Options

- **TargetAdaptor:** specifies target adaptor interface used for connecting to target. Select **mictor38**for the OMAP4460 PandaBoard. The following table shows the supported Target Adaptors:

Target Adaptors	Remarks
arm20	Standard ARM 20 pin JTAG connector (debug only)
mictor38	Standard MICTOR 38 connector(debug and trace). This setting should also be used with the ARM38-CTI20 adapter i.e. when your target has a TI CTI20 adapter e.g. TI Sitara AM335x (TMDXEVM3358**)** board
csdbg	Standard CoreSight 20 pin connector with debug only
csdbgtrace	Standard CoreSight 20 pin connector with debug and trace
cs10	Standard CoreSight 10 pin connector with debug only
mictor38tomipi34	STM adaptor for PandaBoard
stmptmcombo	OMAP5 adaptor with both PTM and STM support

Table 1. Vitra-XD Target Adaptors

- **Device:** specifies the ARM device/core we wish to debug. Select **TI_OMAP4430_A9CORE1 or TI_OMAP4430_A9CORE2** for the OMAP4460 or OMAP4430 PandaBoards.
  Note:TI_OMAP4430 should be selected only in case of SMP process tracing. For debugging bare-metal or non-SMP kernel, either *TI_OMAP4430_A9CORE1* or *TI_OMAP4430_A9CORE2* has to be selected.
- **Cores:**specifies the cores to be debugged (for multi-core systems). The coresare specified as a comma separated list.
- **Endianness:** allows you to specify the memory endianness of your target system which should be **little** for OMAP4460.
- **Voltage:** allows you to tell Vitra-XD the voltage level of the target. Select **auto** for OMAP4460.
- **JTAG Clock:** specifies the JTAG TCK frequency used to communicate with the JTAG interface on your ARM device.20000kHz (20MHz) is the optimum value for OMAP4460.
- **Reset** options include
  - **none**. Sourcery CodeBenchsimply stops the core after configuration.
  - **delay (recommended default option)**. Sourcery CodeBenchissues a hard-reset which is defined as asserting the EmbeddedICE connector's nSRST pin. This should also reset the ARM core (assuming nSRST is connected to your ARM core reset as per ARM's recommendations). After a hard-reset, Sourcery CodeBenchwaits to allow proper target initialization prior to communicating with the target (i.e. entering Debug mode). The delay

prior to target initialization is configurable. An ARM core begins execution from 0x0000 after reset. Typically, this is ROM or flash based code and contains any startup or initialization code. The last thing the startup code normally does is write to the re-map register which remaps RAM to the lower memory from 0x0000 onwards. Sourcery CodeBenchassumes that this initialization has been executed during the delay, and that, when the core is stopped, code can then be downloaded to RAM and execution can begin.

  - **halt.** This option causes Sourcery CodeBenchto issue a hard reset and halt at the Reset Vector. This allows you to begin debugging from the very first instruction in your application.
  - **hotplug (connect to running target).** This option allows you to connect to an already running target. Make sure you have selected the appropriate target **Voltage** for your target and when prompted, connect Vitra-XD to your target board.
  - **core.** This reset option is only available for Cortex-M cores. Vitra-XD sets the VECTRESET bit. The Cortex-M will halt on reset as Vitra-XD enables vector catch on reset.
  - **device.** This reset option is only available for Cortex-M cores. Vitra-XD sets the VECTRESET and *SYSRESETREQ* bits. The Cortex-M will halt on reset as Vitra-XD enables vector catch on reset.

- **ResetDelay:** specifies the delay in seconds after which the core has to be halted following a hard reset.
- **DBGRQ Low:** allows you to control the Debug Request line (DBGRQ) on the EmbeddedICE interface. By default, DBGRQ is set to low, however, certain targets may require this to be set high (for example, DBGRQ may be used to switch between debug and boundary scan modes of operation for the EmbeddedICE (JTAG) interface).
- **Cache Clean Address:** If your ARM device has cache,then Sourcery CodeBench needs to execute a routine after every program halt to ensure cache is invalidated and write-back is performed. This control allows you to specify the location of the invalidation routine which requires 128 bytes. The invalidation routine is supplied with Sourcery CodeBench and automatically downloaded as needed.
- **Safe Address**(only applicable for ARM7 and ARM9 devices; not for Cortex)**:** During debugging, the ARM core PC is incremented while executing ARM core instructions (for example to read/write memory via JTAG using Vitra-XD) meaning that code is pre-fetched from the PC address. Safe non-vector address allows you to specify a valid PC address to prevent code fetches from an invalid memory range in your application (which may cause an exception or fault). When enabled, the PC is set to this safe address before every memory access, hence, the performance impact.
- **Semihosting:** enable/disable semihosting. Semi-hosting is a mechanism whereby the ARM target communicates I/O requests made in the application code, such as `printf()`, `scanf()`, etc., up to the host computer running the debugger, rather than having a screen/keyboard/disk on the target system itself.

11. Select **Source** tab next to **Debugger** tab to map the source files corresponding to the application to be debugged (This is done automatically if you are debugging an application that you build inside CodeBench).

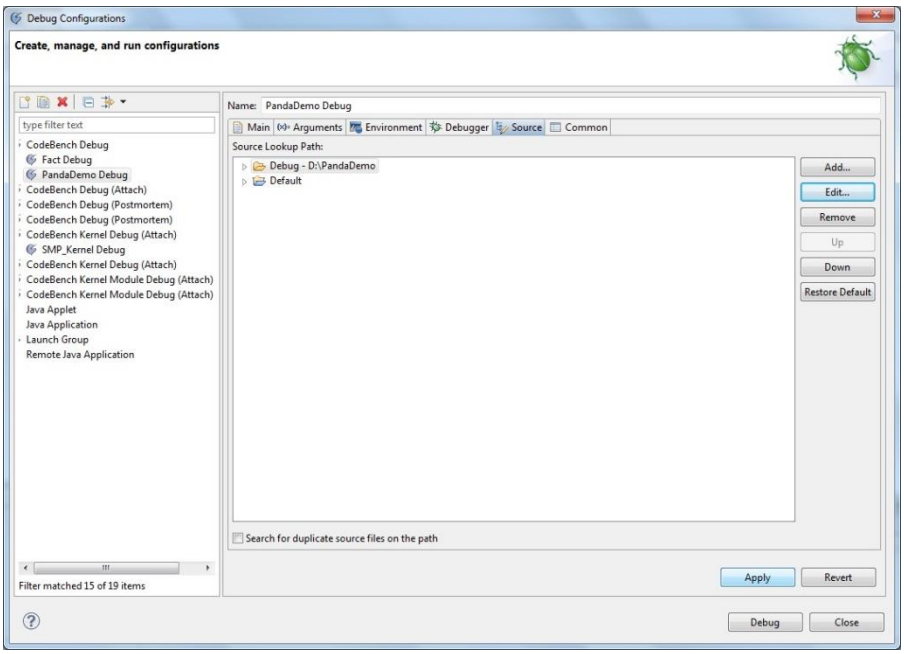


Figure 18. Source mapping

12. Click **Apply** and **Debug** which will download the application to the target.

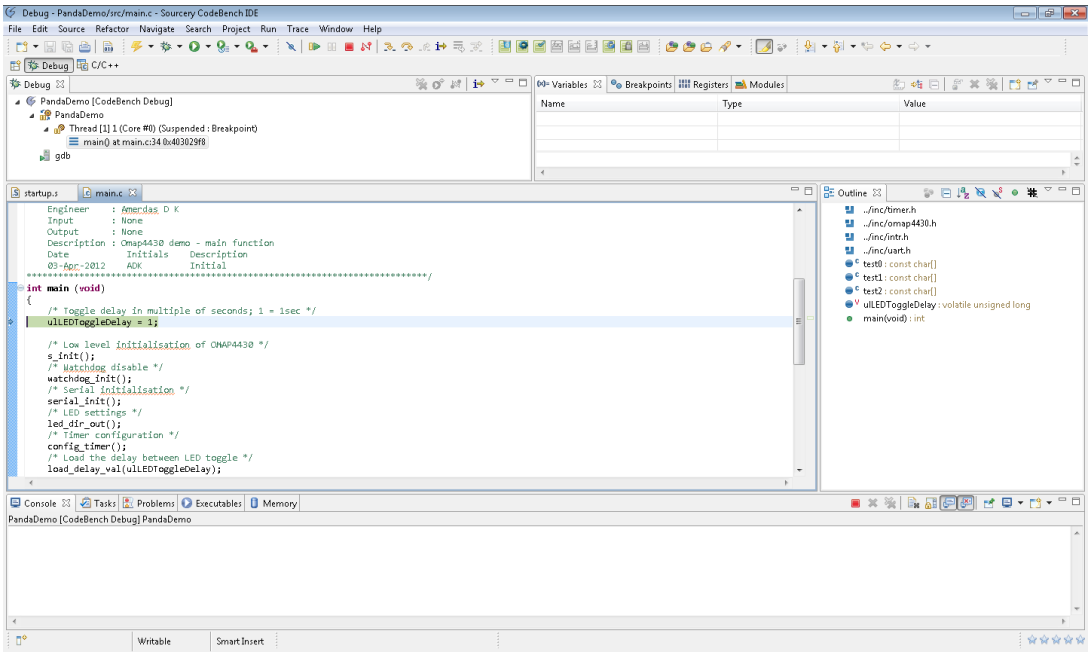


Figure 19. Debug perspective

After downloading, the application will halt at main function.

4. Trace support

When used in conjunction with Vitra-XD, Sourcery CodeBench supports real-time capture, reconstruction and display of code trace. This is based on the bus trace information emitted from the target's trace port (e.g. Program Trace Macro-cell(PTM)for the OMAP4430). The emitted trace data is captured by Vitra-XD, time stamped, and transferred to the host PC. Sourcery CodeBench will reconstruct program flow based on the raw trace data, and display it in a readable format in its custom trace views.

Setting up a trace session in Sourcery CodeBench is a 3-step process:

i. Enabling target trace pins
ii. Configure trace
iii. Autolock(optional)

4.1 Enabling target trace pins

IMPORTANT: Some devices multiplex the trace pins with other functionality (e.g. IO) and hence the pins must be configured to work in trace mode. This can be done using a GDB script which is executed by CodeBench during the target connection process. Sample GDB scripts are provided for OMAP44xx, OMAP54xx, AM335x, Cortex-M3 and i.MX6 targets and are available in the following directories (Windows hosts):

- *<Installation_path\codebench\i686-mingw32\arm-none-eabi\ash\scripts>*
- *<Installation_path\codebench\i686-mingw32\arm-none-linux-gnueabi\ash\scripts>*

or the following directories (Linux hosts):

- *<Installation_path\codebench\i686-pc-linux-gnu\arm-none-eabi\ash\scripts>*
- *<Installation_path\codebench\i686-pc-linux-gnu\arm-none-linux-gnueabi\ash\scripts>*

The below screen-shot shows how the script is invoked within CodeBench for configuring the target trace pins (via **GDB cmd**)using GDB's *source* command as shown below (don't forget to prefix with *source)*:

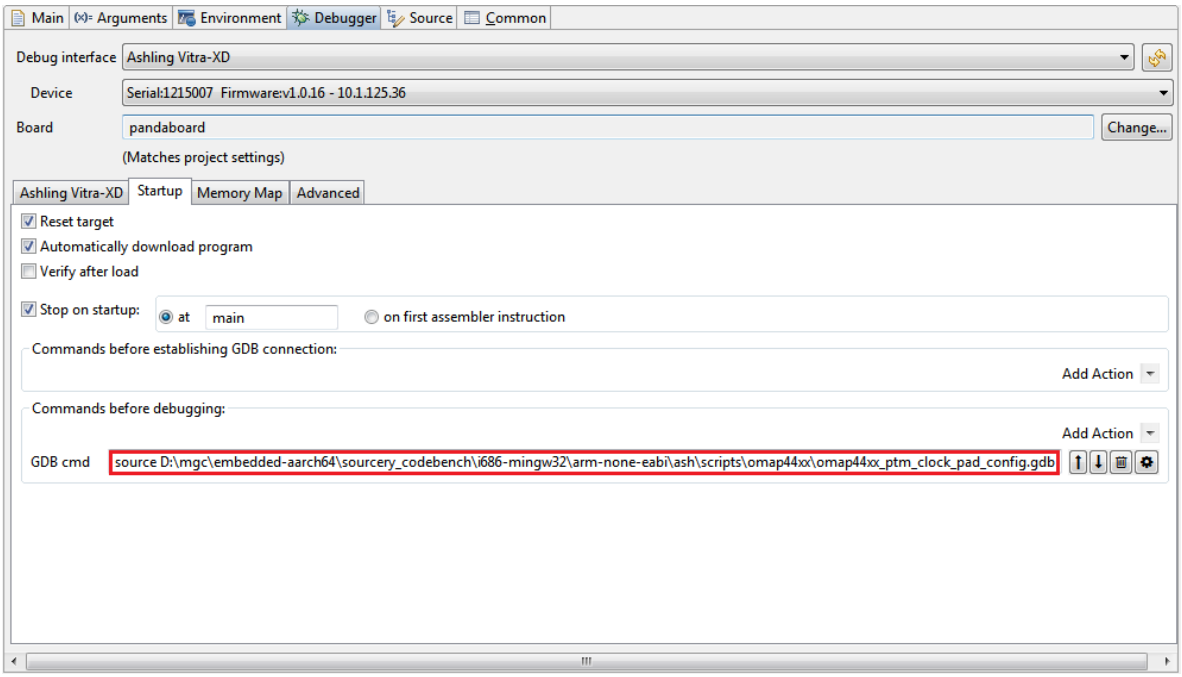


Figure 20. Configuring OMAP4430 hardware pins for PTM trace using GDB script

This is a vital step and tracing will not work without it, resulting in the following error message:

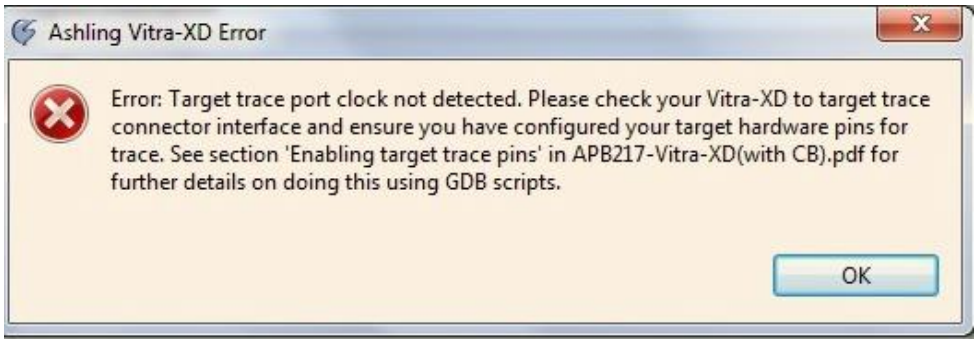


Figure 21. Error message if trace is not configured properly

4.2 Configure trace

To configure trace, invoke the trace configuration dialog from **Trace|Advanced Trace Configuration** menu.

The **Advanced Trace Configuration** dialog is organised into Vitra-XD configuration tab, Protocol specific tabs and Trigger/Trace filtering tab. There can be three or more tabs depending upon the trace protocols supported by the device, as shown in the following screenshot:

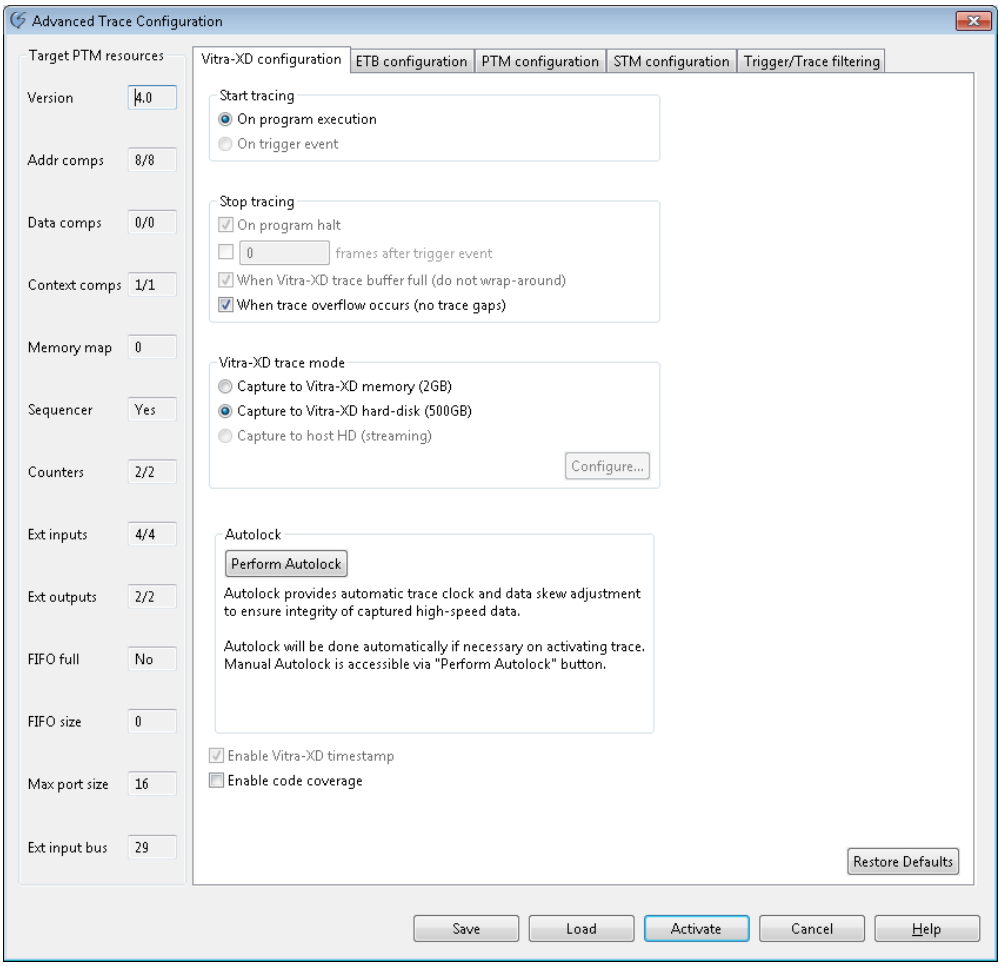


Figure 22. Trace configuration dialog - Vitra-XD configuration

Vitra-XD configuration options include:

- **Start tracing**
  - **On program execution.** Trace data will be captured right from program execution
  - **On trigger event.** Trace data will be captured only when a trigger event occurs (This option is enabled only if the target supports trigger. OMAP4460/4430does not support trigger)
- **Stop tracing**
  - **On program halt.** Trace capturing stops when program execution halts

- **<x> frames after trigger event.** Trace capturing stops <x> frames after the occurrence of a trigger event
- **When Vitra-XD trace buffer full (do not wrap-around).**Trace capturing stops when the Vitra-XD memory/hard-disk becomes full
- **When trace overflow occurs (no trace gaps).**This option is applicable in the case of ***Capture to Vitra-XD hard-disk*** mode only. Trace capturing stops when Vitra-XD memory becomes full ('trace overflow'_ as it means that Vitra-XD is not able to write trace data to its hard-disk fast enough (and thus will 'lose' trace information). In practise this should not happen.

- **Vitra-XD trace mode**
  - **Capture to Vitra-XD memory (2GB).**Trace data is captured to Vitra-XD's on-board high-speed DDR3 memory (2GB) and trace data can only be viewed when the target ARM core is halted.
  - **Capture to Vitra-XD hard-disk (500GB).**Trace data is captured to Vitra-XD's on-board 500GB hard-disk (utilizing the 2GB DDR3 memory as a FIFO buffer). Trace data can be viewed when the target ARM core is running.
- **Autolock. See the following section** Autolock.
- **Enable code coverage.** See the following section on Code coverage.

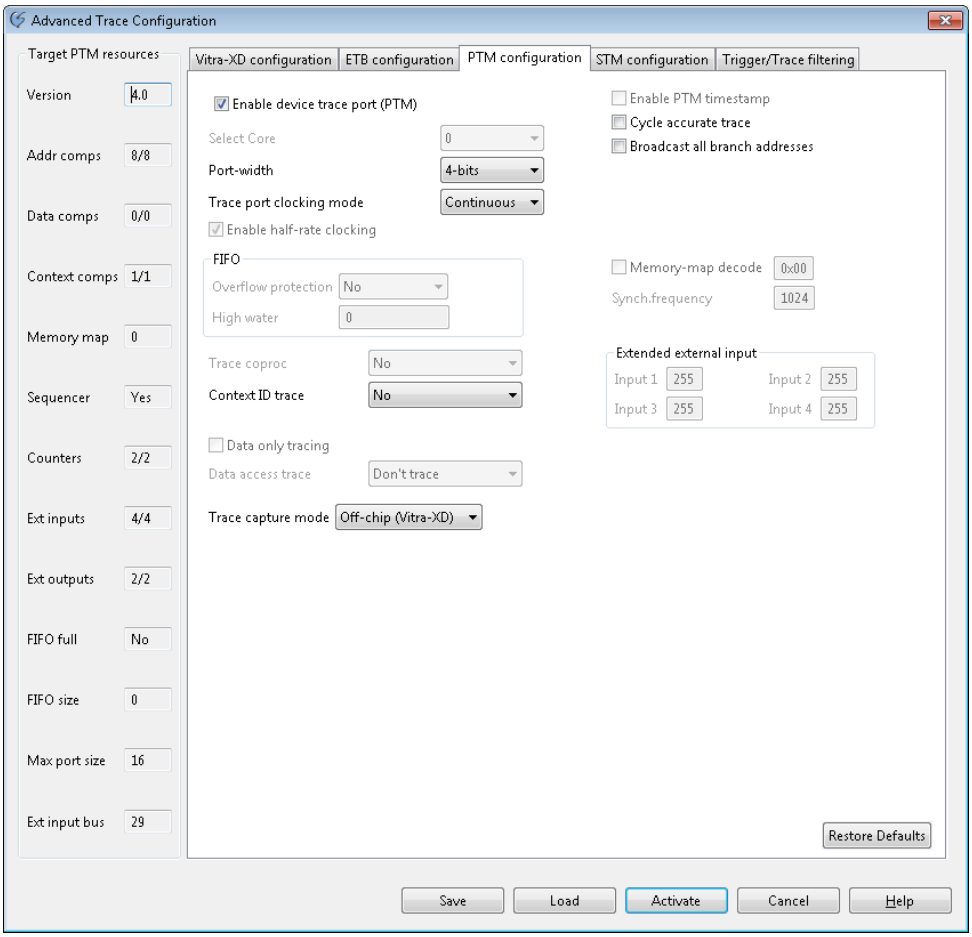


Figure 23. Trace configuration dialog - ETM/PTM configuration

PTM configuration configures the target's trace port and options include:

- **Port-width.** Target's trace port width can be configured. Supported widths are**4-bits**, **8-bits**and**16-bits**
- **Trace port clocking mode.** Target's trace port mode as per ETM/PTM specification.
  - **Normal** mode. Decreases trace port bandwidth requirements by using a compressed packet format.
  - **Bypass** mode. Non-compressed packet format.
  - **Continuous** mode**.** Compressed packet format(does not use the TRACECTL signal)
- **Cycle accurate trace.** If this option is checked, then information on the number of cycles taken for executing each instruction is embedded in the trace data emitted by the target. This includes cycles during which no trace information is normally returned, such as memory wait states. Sourcery CodeBench decodes the cycle count information embedded in the trace data and displays it in readable format in the trace view.
- **Broadcast all branch addresses.** If this option is checked, then the PTM outputs the destination address for all branch instructions, including direct branch instructions. This is useful for tracing parts of a program where you do not have a code image, for example for tracing an area of self-modifying code.

Trigger/Trace filtering configures the trigger and filter conditions for trace capturing.

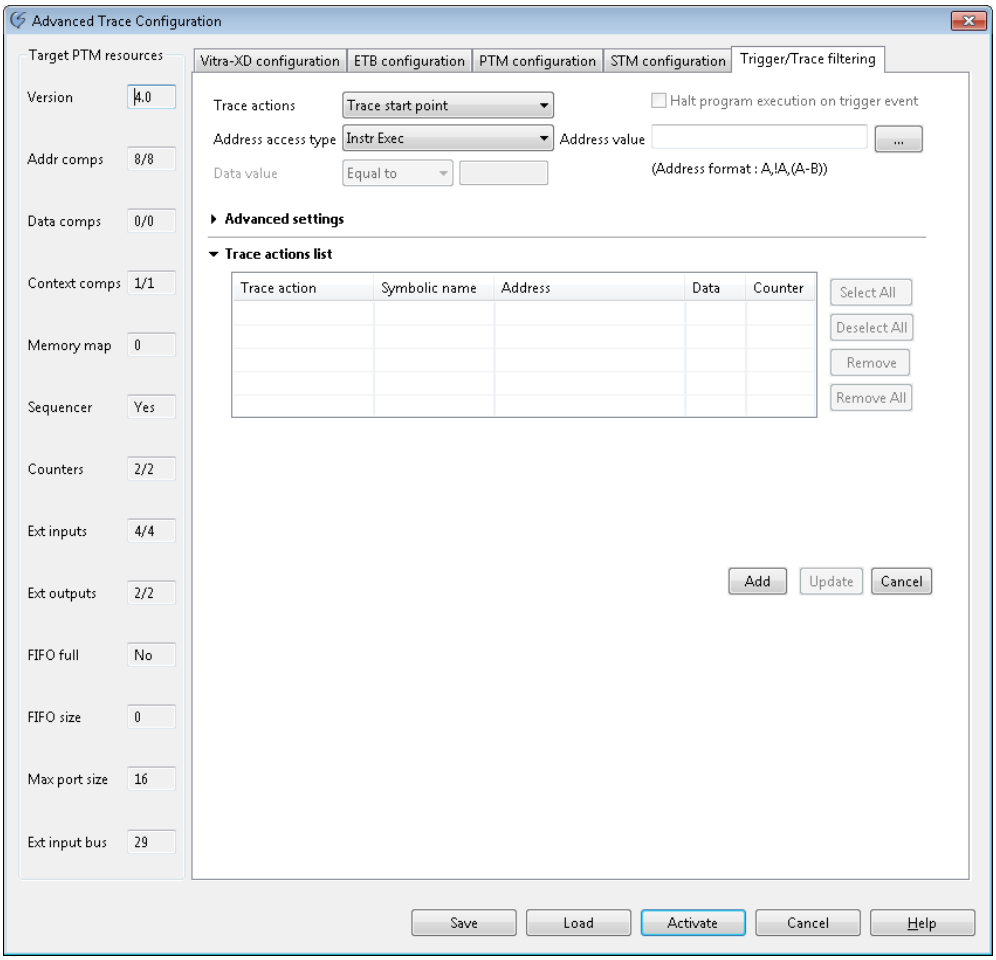


Figure 24. Trace configuration dialog - Trigger/Trace filtering

Trigger/Trace filtering options include

- **Trace actions**
  - **Trigger.** Specifies the point at which the on-board trace macro cell (e.g. ETM or PTM) will emit a trigger event. Trigger events are monitored by Vitra-XD and can be used to start and stop tracing (see **Vitra-XD configuration** tab).
  - **Trace start point.** Specifies the point at which the target will sta**rt** emitting trace data. To add a **Trace start point**, select it from **Trace actions** drop-down, and then browse from **Address value** edit box, the address of the function you would like to start the trace. Once selected, click the **Add** button. The selected trace action will then be shown in the **Trace actions list**.
  - **Trace stop point.** Specifies the point at which the target will stop emitting trace data. To add a **Trace stop point**, select it from **Trace actions** drop-down, and then browse from **Address value** edit box, the address of the function at which you would like to stop the trace. Once selected, click the **Add** button. The selected trace action will then be shown in the **Trace actions list**.

    *Note: ARM defines a waypoint as an indirect branch, conditional branch, direct branch or an exception. When using devices based on PTM, a **Trace start point** or **Trace stop point** will not be activated (and thus start/stop emitting trace data) until a waypoint is executed on or after the Trace start/stop point address.*

  - **Trace include.** Specifies the range of addresses between which the target will emit trace data(both addresses inclusive). To add a **Trace include**, select it from **Trace actions** drop-down, and then browse from **Address value** edit box, the addresses of the functions you would like to include for tracing. Once selected, click the **Add** button. The selected trace action will then be shown in the **Trace actions list**.
  - **Trace exclude.** Specifies the range of addresses between which the target will not emit trace data. To add a **Trace exclude**, select it from **Trace actions** drop-down, and then browse from **Address value** edit box, the addresses of the functions you would like to exclude for tracing. Once selected, click the **Add** button. The selected trace action will then be shown in the **Trace actions list**.

- **Exclude kernel trace.** This option allows exclusion of Linux kernel tracing i.e. trace processes only. This option is available only if the tool chain selected in the project configuration is Sourcery CodeBench for ARM GNU/Linux. .
- **Trace a specific process.** Allows tracing of a specific Linux process by specifying its Process ID (PID). This option is available only if the tool chain selected in the project configuration is Sourcery CodeBench for ARM GNU/Linux.

- **Address access type.** Specifies the condition for trace start/stop. This field is enabled based on target support. Available options are:
  - Instr Exec – Match only when the *instruction at the given address* reaches the *Execute* stage of the pipeline.
  - Instr Fetch– Match only when the *instruction at the given address* reaches the *Fetch* stage of the pipeline.
  - Instr Exec and pass condition – Match only when the *instruction at the given address* reaches the *Execute* stage of the pipeline and passes the *condition code*(e.g.: GT, LT,HI, LS etc).
  - Instr Exec and fail condition– Match only when the *instruction at the given address* reaches the *Execute* stage of the pipeline and fails the *condition code* (e.g.: GT, LT,HE, LS etc) .
  - Data read – Match only when a *read* operation is performed at the given address.
  - Data write – Match only when a *write* operation is performed at the given address.
  - Data access – Match when either read or write operation is performed at the given address.

NOTE: The TI OMAP4460/4430 device supports only *Instr Exec* option.

- **Data value.** Specifies how data is to be traced if the target supports data tracing. This option is enabled based on target support. Available options are:
  - Equal to
  - Not equal to

NOTE: The TI OMAP4460/4430 device does not support data tracing.

4.2.1 Setting Trace actions from the Source view

Trace actions can be set from the Source view by right-clicking on the ruler and selecting the appropriate options from the Trace context menu. The context menu items will be enabled only if the selected source line is executable.

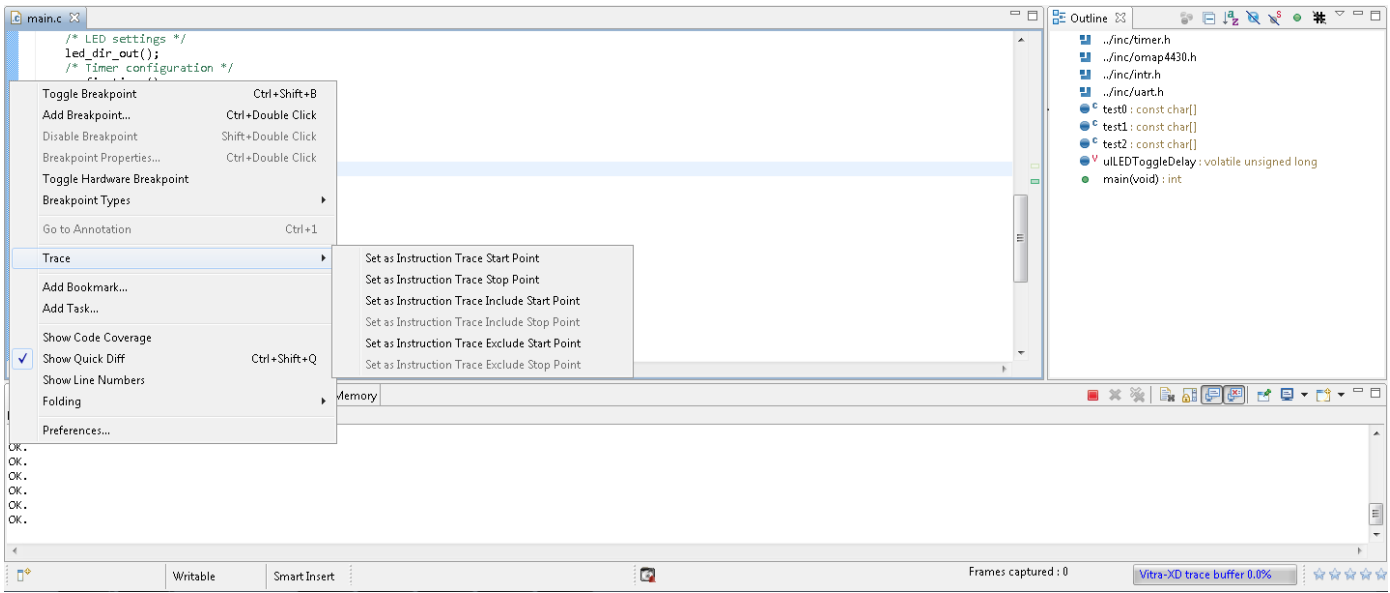


Figure 25.Trace context menu for Trace actions

In the Source view trace markers corresponding to the trace actions will be displayed in the ruler as shown below:.

```
33      /* Toggle delay in multiple of seconds; 1 = 1sec */
34      ulLEDToggleDelay = 10;
35
36      /* Low level initialisation of OMAP4430 */
37      s_init();
38      /* Watchdog disable */
39      watchdog_init();
40      /* Serial initialisation */
41      serial_init();
42      /* LED settings */
43      led_dir_out();
44      /* Timer configuration */
45      config_timer();
46      /* Load the delay between LED toggle */
47      load_delay_val(ulLEDToggleDelay);
48      /* Global interrupt enable - 39 */
49      enable_interrupts(GPT3_INT);
```

Figure 26: Editor with trace markers in the ruler

 - Trace Start point

 - Trace Include start point

 - Trace Exclude start point

 - Trace Stop point

 - Trace Include stop point

 - Trace Exclude stop point

4.3 STM Trace configuration

Vitra-XD also supports the System Trace Module (STM) as used in TI's OMAP4/5 or Sitara families. STM provides software developers with a hardware-accelerated, multi-core "printf" ability. In a multi-core environment, messages from each core are identified and globally time stamped by hardware before being emitted via the devices trace port to be captured by Vitra-XD. STM messages are shown in a dedicated view in the Sourcery CodeBench IDE. STM support requires the **mictor38tomipi34**target adaptor for OMAP boards (Sitara requires the ARM38-CTI20 adapter).

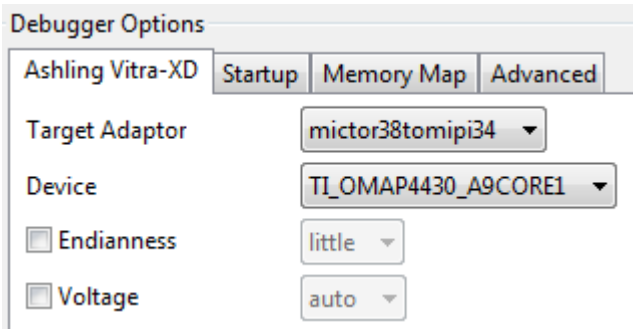


Figure 27. Selecting the appropriate Target adaptor for STM configuration

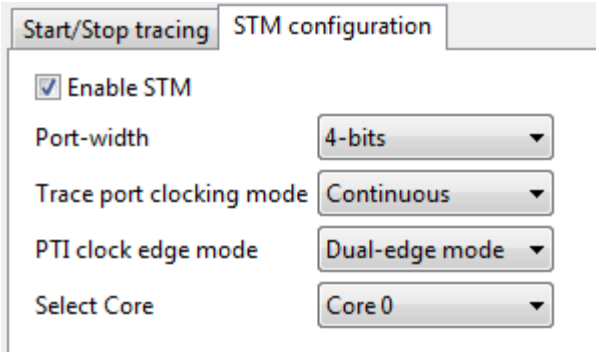


Figure 28. STM Configuration

- **Port-width.** Target's STM trace port width.

- **Trace port clocking mode.** Specifies if the target trace clock has to be generated when the target FIFO is empty. Currently, only **Continuous** mode is supported.
- **PTI clock edge mode.** Indicates the clock edge where valid data is emitted i.e. on the rising edges only or on both rising and falling edges (**Dual-edge mode)**.
- **Select Core.** Used to select the MPU subsystems to be used as software master.

4.4 ETB (On-chip) Trace configuration

The ETB (on-chip) trace feature makes use of the *Embedded Trace Buffer* (*ETB*) available on certain ARM based processors. The ETB configuration parameters are available in a separate tab called *ETB configuration* tab in *Advanced Trace Configuration* dialog. This tab will be displayed only if the connected device has *ETB* support.

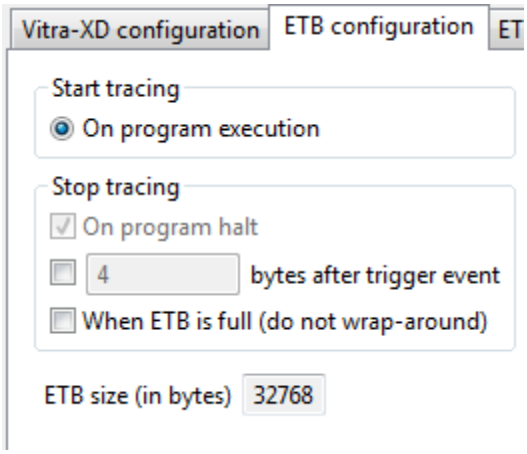


Figure 29. ETB Configuration

ETB configuration options include:

- **Start tracing**
  - **On program execution.** Trace will be captured in *ETB* when the core starts running.
- **Stop tracing**
  - **On program halt.** Trace capturing will stop when the core is halted. This will be checked and greyed out to indicate that irrespective of other trace settings, trace capture will stop when the core is halted.
  - **N bytes after trigger event.** Stop trace capturing after a trigger event has occurred. *N* should be less than the total buffer size displayed in *ETB size*.
  - **When ETB is full (do not wrap around).** If checked, trace capturing will stop when *ETB* is full. When unchecked, trace capturing will not stop even after *ETB* is full, and the ETB will work as a FIFO
- **ETB size**
  - This indicates the total size of *ETB*(*in bytes*), read from the target.

4.5 Simultaneous STM and ETB

Trace capture mode enables the user to select whether the trace has to be captured *off-chip* via *Vitra-XD* probe or *on-chip via ETB*. This shall be available in each of the ETM/PTM/STM configuration tabs thereby allowing independent configurations; e.g. PTM via on-chip and STM via off-chip on the TI AM335x Sitara device.

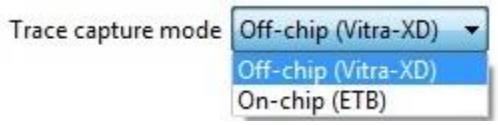


Figure 30. Trace capture mode option in ETM/PTM/STM configuration tab

4.6 Autolock

High-speed trace data is routed from your ARM device pins to the trace connector on your target system. Due to PCB tracking issues etc., there may be skews between trace data and clock and also among the individual trace data lines. These can cause setup/hold time violations and reduced eye width of trace data, which in turn can corrupt trace data being captured by Vitra-XD. To overcome this problem, Vitra-XD provides a mechanism named *Autolock* which allows automatic skew adjustment of trace data/clock lines to provide better integrity of parallel trace data captured. *Autolock* can be initiated from **Trace|Autolock** menu.

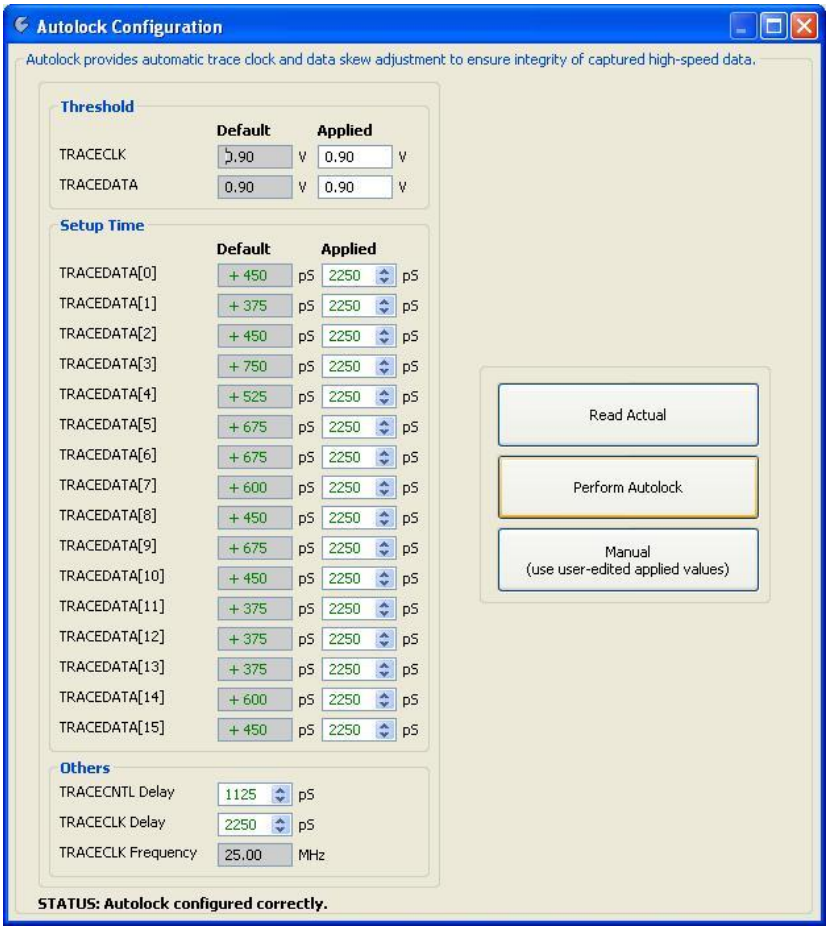


Figure 31.Autolock UI in Sourcery CodeBench

- **Threshold** fields show the voltage thresholds used in the signal conditioning circuits for trace clock and data lines.
- **Setup Time** fields show the **Default**(before Autolock) and **Applied** (after Autolock/Manual) setup times of each trace data line with respect to the trace clock.
- **Others** fields show the delay value applied to trace control (**TRACECNTL**), the delay value applied to trace clock (**TRACECLK)**, and the frequency of trace clock (**TRACECLK Frequency**).

The following actions are available:

- **Read Actual**. Reads the actual setup times of each trace data line (TRACEDATA[n]) with respect to the trace clock and shows the results in the **Default** fields.
- **Perform Autolock**. Initiates Autolock and shows the applied setup times for each trace data line (with respect to the trace clock) in the **Applied** fields. **TRACECLK Frequency** shows the measured trace clock value.
- **Manual.** Applies the user specified/adjusted **Applied, TRACECNTL Delay**(trace control) and **TRACECLK Delay**(trace clock) values. Values can be adjusted in steps of 75pS.

4.7 Viewing trace

Once a trace configuration is activated and the core is run, Vitra-XD starts capturing the trace data emitted by the target. Status is shown in the status bar as follows:

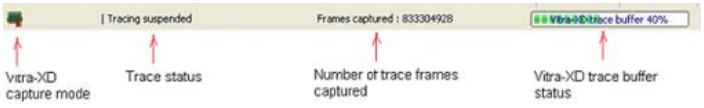


Figure 32. Sourcery CodeBench status bar in off-chip mode

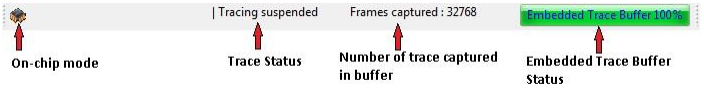


Figure 33. Sourcery CodeBench status bar in on-chip mode

Once captured, trace may be viewed in Sourcery CodeBench using the Trace Perspective from **Trace | View Trace** as shown below:

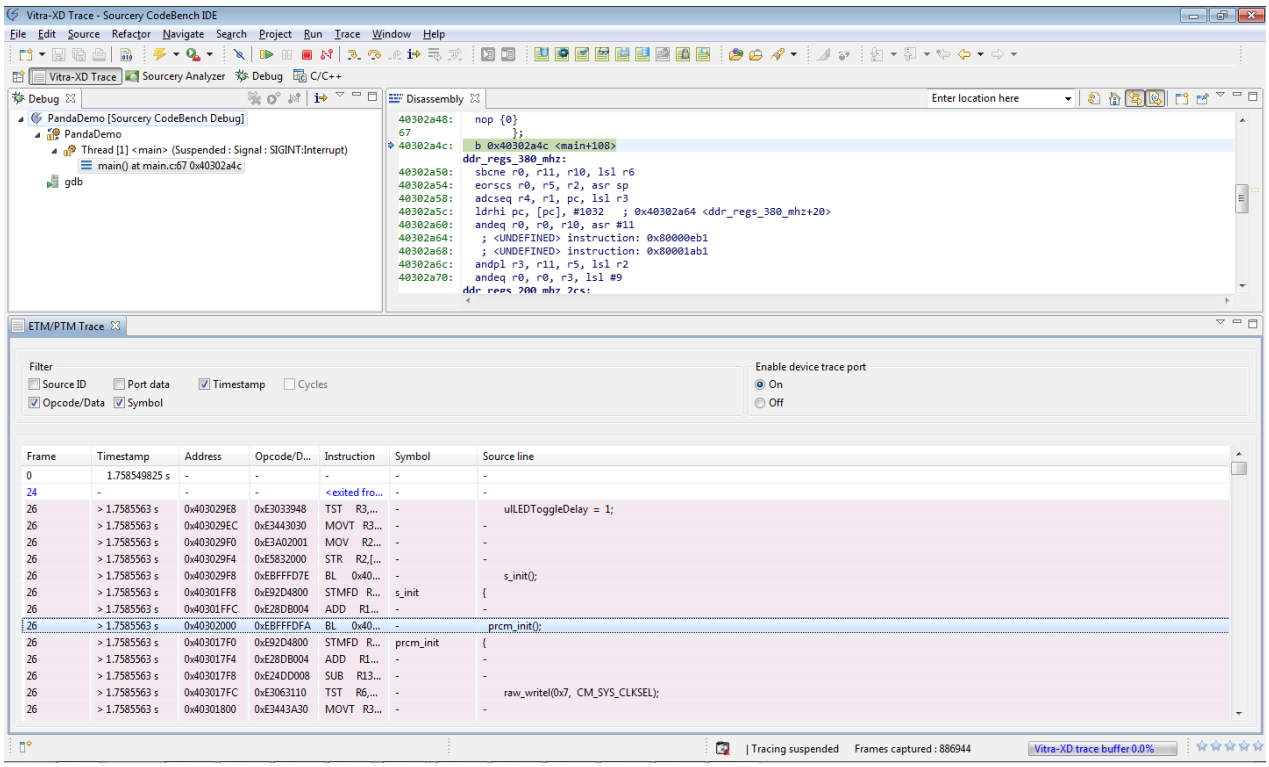


Figure 34. Sourcery CodeBench Trace Perspective

NOTE: In **Capture to Vitra-XD hard-disk** mode, trace can be viewed even when the core is running; in **Capture to Vitra-XD memory** mode, trace can only be viewed only after halting the core.

Trace information can be selectively filtered (i.e. removed from the window) via the **Filter** control. For example, checking **Port Data** will show the raw data emitted from target's trace port as shown below:

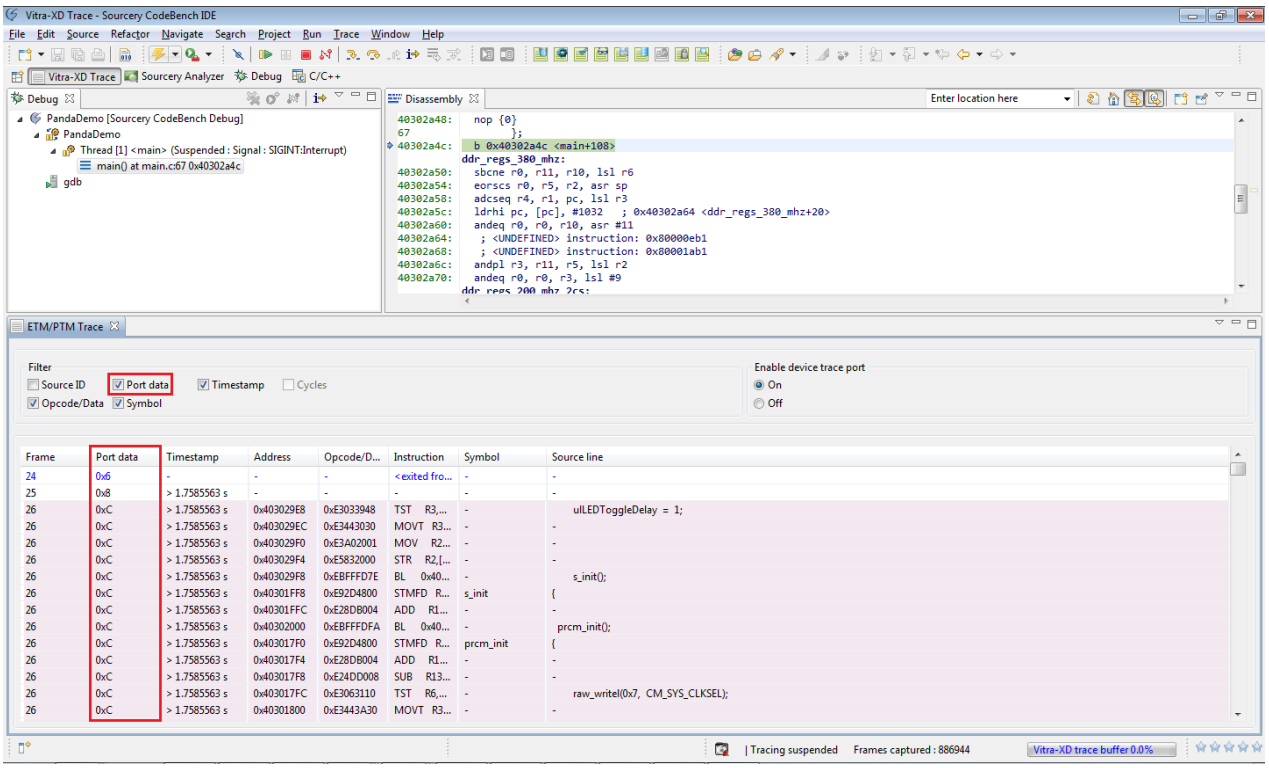


Figure 35. Trace view with Port Data checked

Show process trace and ***Show kernel trace*** allow filtering of Linux process and kernel information. ***Filter processes*** brings up a dialog which allows specification of specific processes for filtering.

STM Trace view will look like below:

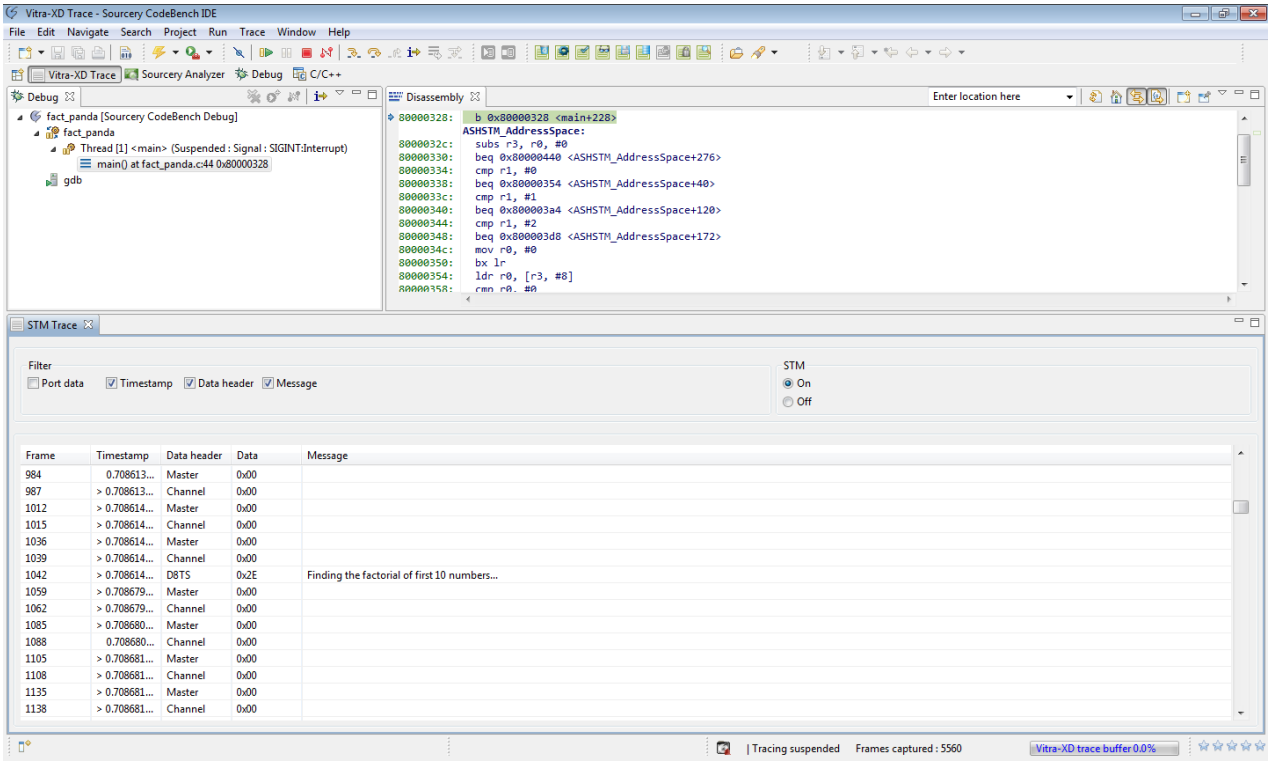


Figure 36. STM Trace view

Sourcery CodeBench provides a Trace control bar where you can access the trace features as follows:

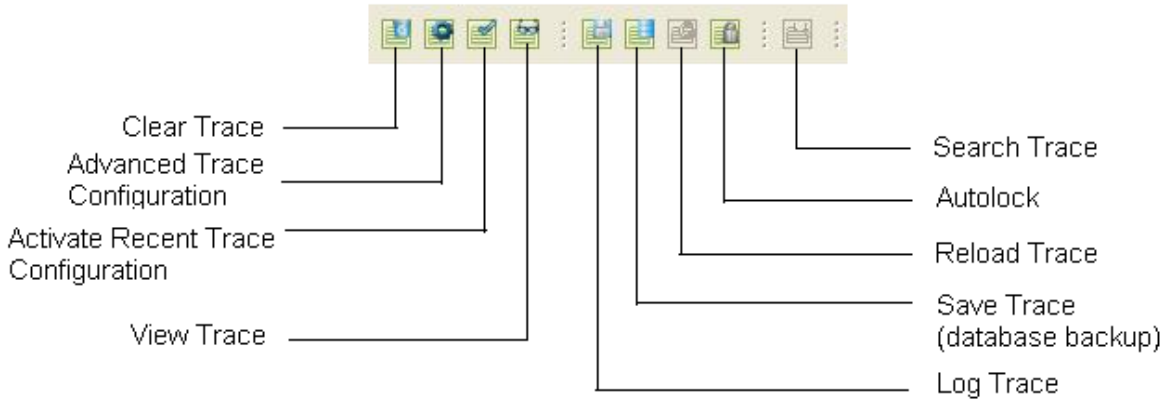


Figure 37. Sourcery CodeBench Trace Control bar

The options provided in Trace control bar include

- **Clear Trace.** Clears the entire captured trace information. Captured data is appended to the trace buffer, hence, clear it before a new session if you do not want to save the older trace data.
- **Advanced Trace Configuration.** Configure a new trace session (brings up **Trace Configuration** dialog)
- **Activate Recent Trace Configuration.** Activates the most recent trace configuration
- **View Trace.** Opens the Sourcery CodeBenchtrace perspective allowing viewing of captured trace data
- **Search Trace.** Search within the captured trace frames. Explained in section **4.8** .
- **Autolock.** Pops-up the Autolock dialog for performing Autolock.
- **Reload Trace.** Reloads a previously saved trace session (using **Save Trace)**. Only available when **Enable Trace** is unchecked.
- **Save Trace.** Save captured trace data in SQL(.DBO) format. This can be reloaded into Sourcery CodeBench for viewing later.
- **Log Trace.** Save the captured trace data in .CSV and .TXT formats

Sourcery CodeBench also supports *trace tracking*; i.e., double-clicking any line in the **Trace** view will show the corresponding source line in the **Source** view.

4.8 Search trace

Sourcery CodeBench allows you to search trace frames captured via the **Trace|Search Trace Data** menu

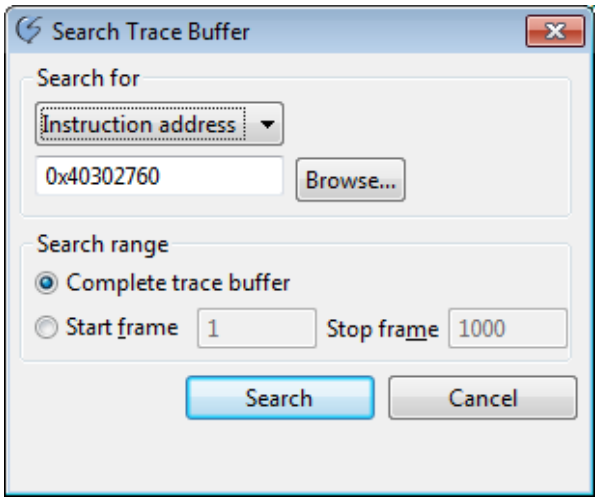


Figure 38. Search Trace

Browse for a symbol address and click **Search**. You can see the Search progress bar as below.

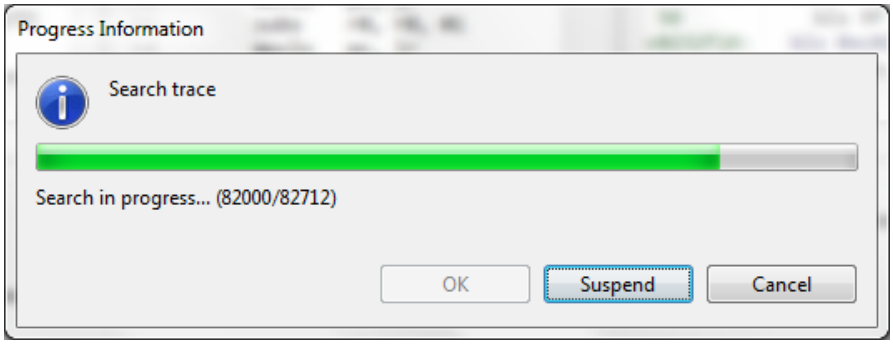


Figure 39. Search trace progress

Once the search completes, you can see the search results on the right side of the Sourcery CodeBench trace perspective. Double clicking the search results will display the corresponding frames in the **Trace** view as shown below:

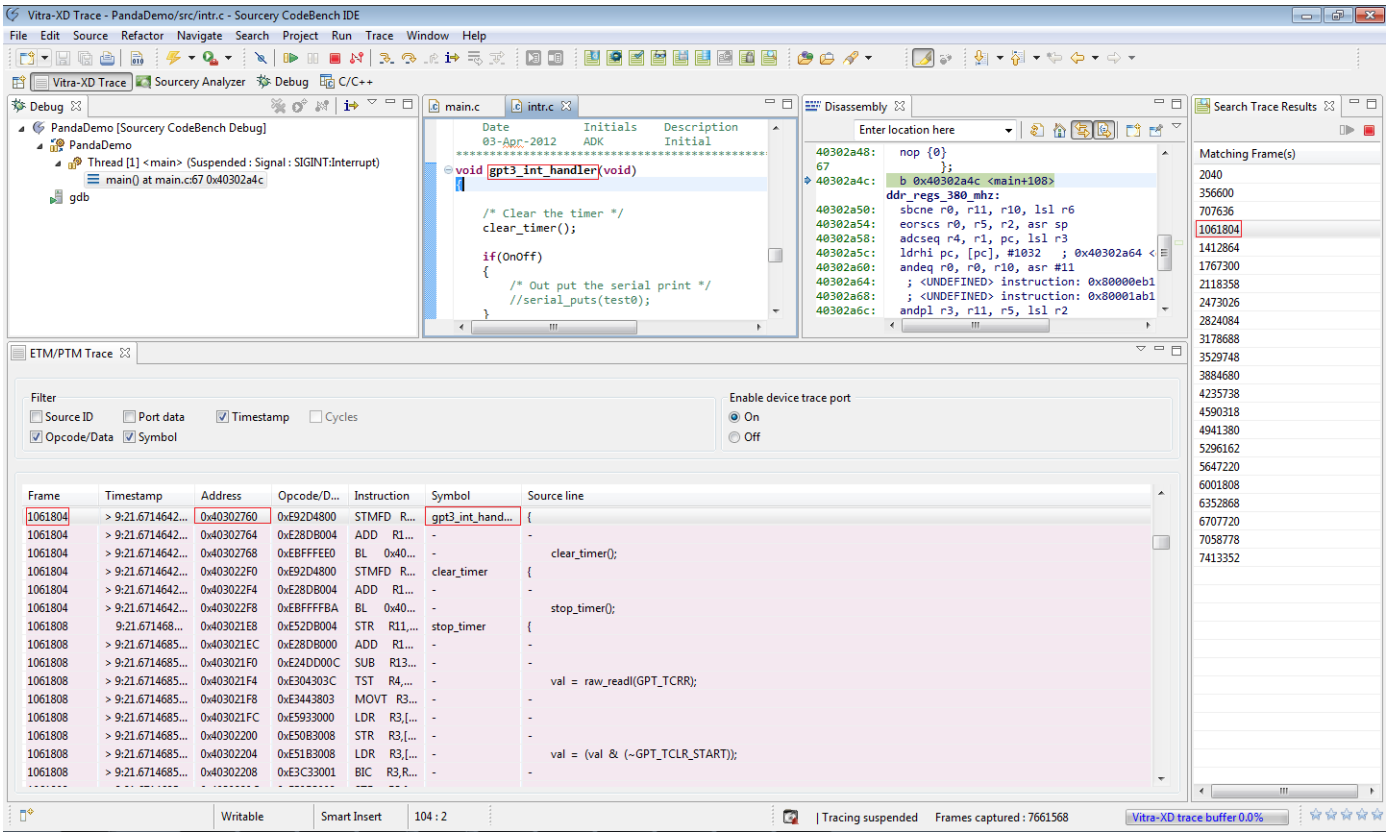


Figure 40. Search trace results

4.9 Saving/Logging trace

Sourcery CodeBench allows you to log/save the captured trace data in the following formats:

i. .CSV Suitable for viewing as a spread-sheet e.g. with Excel
ii. .TXT ASCII text file
iii. .DBO SQL format (can be reloaded into Sourcery CodeBench for viewing (trace must be disabled to reload))

In addition, it also supports Raw data format (i.e. the unreconstructed ETM/PTM data) as emitted by the targets trace port. Please refer to **Appendix C. RAW Trace Format** for more details.

Trace data can be saved using **Trace|Save Trace Data(Backup)**.

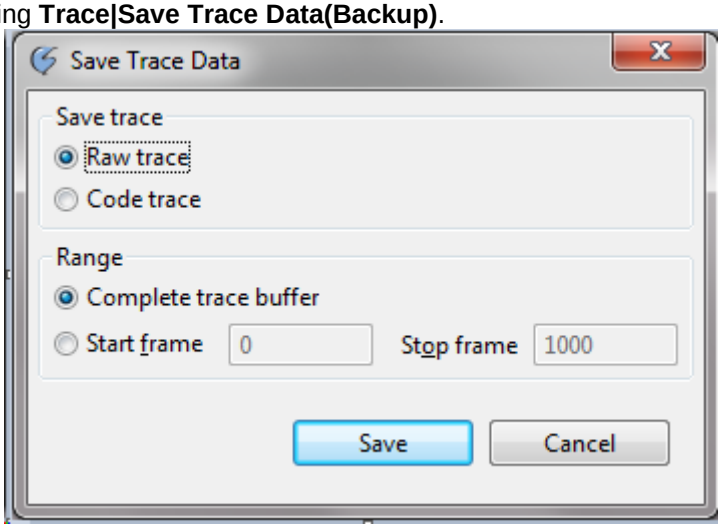


Figure 41: Save Trace Data

Logging trace data to .CSV/.TXT formats can be done via **Trace|Log Trace Data**:

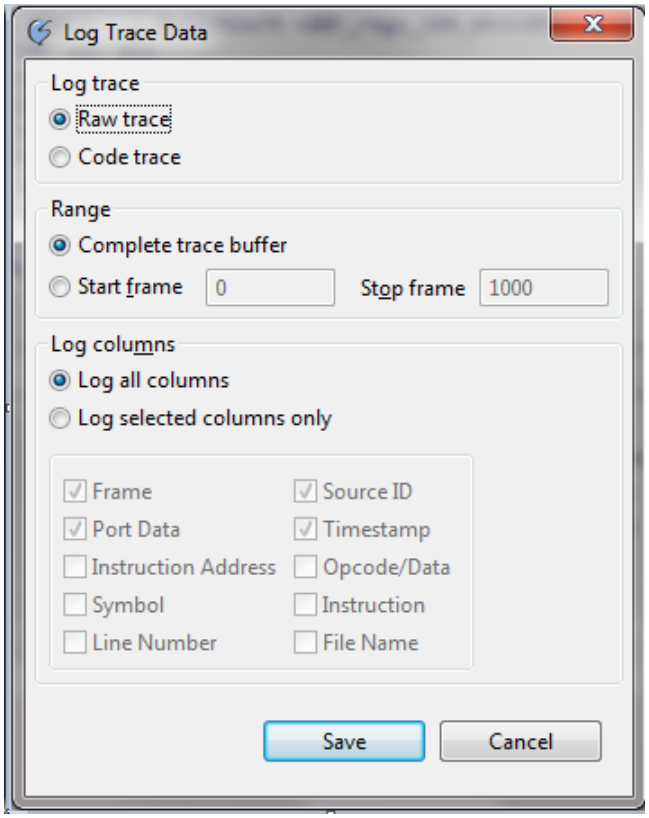


Figure 42. Log Trace Data

4.10 Code coverage

Vitra-XD records and displays the code memory addresses which are accessed during program execution, enabling you to determine which sections of your code have been executed fully (C) or partially (P),and which sections have not. Code coverage can be enabled by selecting the 'Enable code coverage' check-box in the Vitra-XD configuration tab:

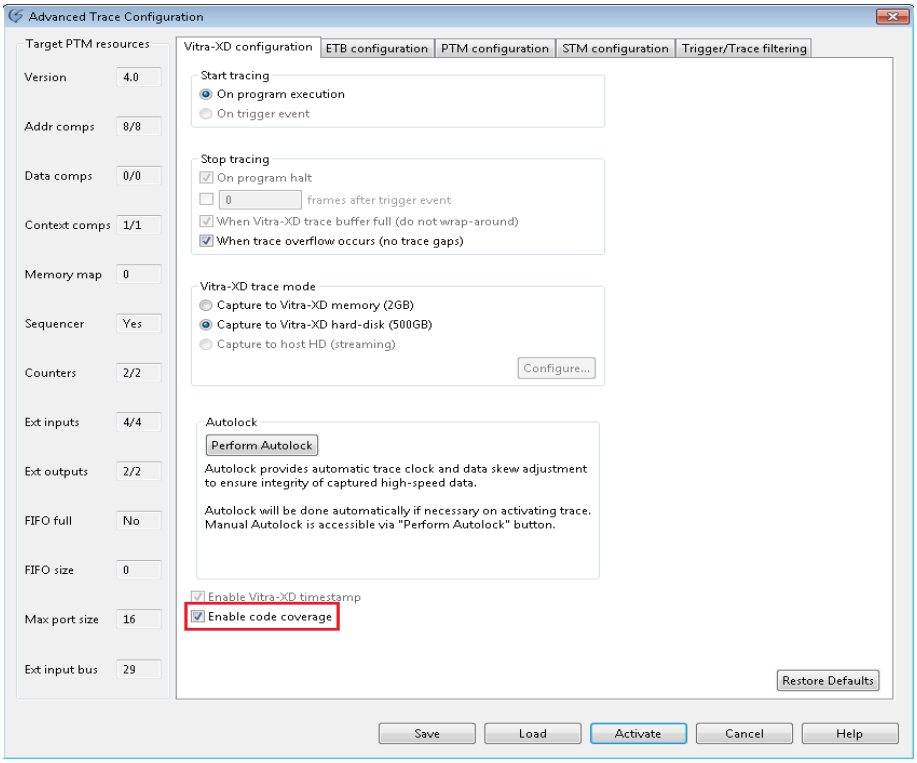


Figure 43. Enabling code coverage

Code Coverage information shall be shown in the Source editor, as in the below screenshot:

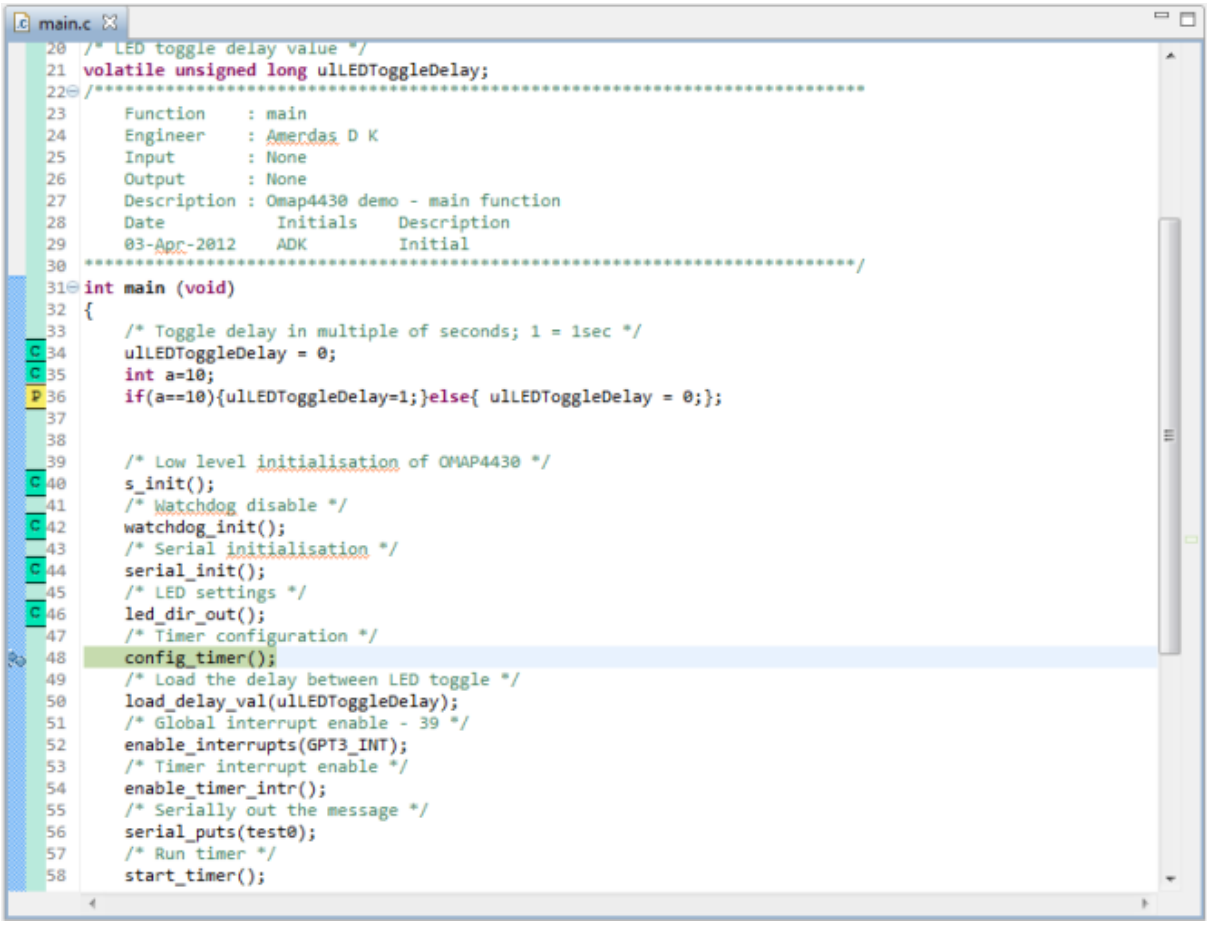


Figure 44. Code coverage view

5. Embedded Linux Debugging with Sourcery CodeBench and Vitra-XD

This section shows how to use Vitra-XD and Sourcery CodeBench to debug and trace Embedded Linux powered targets. A TI OMAP4460 based PandaBoard is used as the target system here.

5.1 Setup

5.1.1 Hardware setup

The following hardware setup is used:

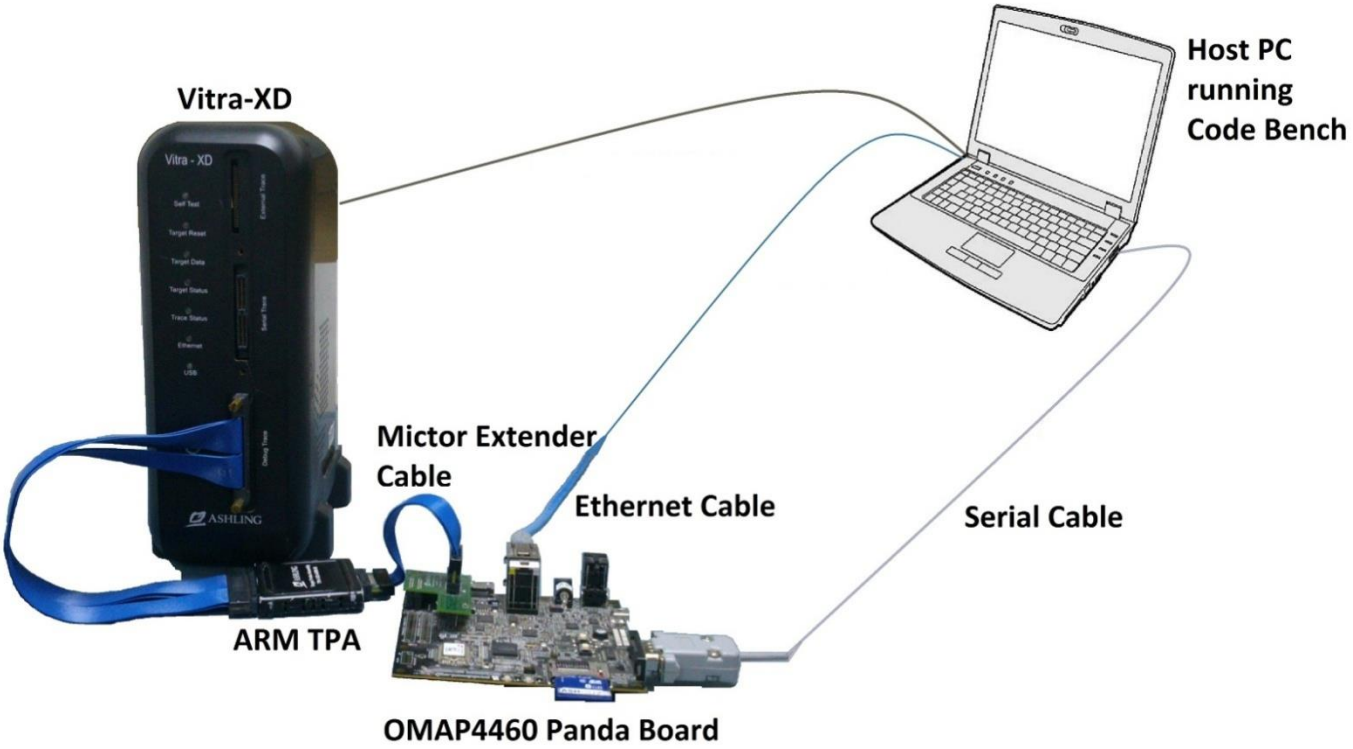


Figure 45. Vitra-XD connected to the TI OMAP4460 PandaBoard

Note the following requirements:

1. The PandaBoard requires an SD card with a boot-loader, Linux kernel image and root file system (ext2). Refer to the Mentor Graphics document *mel_kit_PandaBoard.pdf*(part of the Linux kit for the PandaBoard - http://www.mentor.com/embedded-software/downloads/linux-kits/) for more details.
2. A serial cable is connected between the PandaBoard and the Host PC. This allows running of a Linux shell on the host PC.
3. The PandaBoard must be connected to your host PC via ethernet to support a fast SSH connection to the target file system and run-mode debugging of Linux processes. Use a direct ethernet cable (cross-over) between the PandaBoard and the host PC or alternatively, you can connect your PandaBoard directly to your network using a straight-through cable.

5.1.2 Configuring the PandaBoard IP address

1. If PandaBoard is connected to a DHCP compliant network then it will automatically acquire an IP address from the DHCP server on boot-up.
2. When connected directly to your host PC using a cross-over Ethernet cable then you must configure the PandaBoard's IP address. For example, if your host PC's IP address is set to 192.168.1.2 with netmask 255.255.255.0,then set the IP address of the PandaBoard to192.168.1.1 with netmask 255.255.255.0 using the following command in the PandaBoard shell:
   `>ifconfig usb1 192.168.1.1 netmask 255.255.255.0`
   Note that *usb1* is the Ethernet interface name as displayed via the *ifconfig* command.

5.1.2.1 Configuring the PandaBoard sysroot

For debugging and tracing of shared libraries (both standard libraries as well as custom libraries), sysroot has to be properly configured in the target file system as well as in the host PC. The following steps setup a
a minimal sysroot on the PandaBoard SD card:

1. Create a folder named *libc* in the target file system.
   `>mkdir /libc`
   Note that the user name to login to the target Linux shell is *root* without any password.

2. Copy *<Installation_path\codebench\arm-none-linux-gnueabi\packages\sysroot-minimal.tar.bz2>*to */libc* folder in the target file system using any FTP client (e.g. Filezilla) or via SSH.
3. Change to the directory */libc* in the PandaBoard shell.
   >cd /libc
4. Unzip *sysroot-minimal.tar.bz2*.
   >tar xjf sysroot-minimal.tar.bz2
5. Two folders *lib* and *usr* will now be created after unzipping.
6. Remove the *sysroot-minimal.tar.bz2* file from *libc*
   >rm sysroot-minimal.tar.bz2

5.2 Non-SMP Kernel Debug and Tracing

To debug and trace Linux kernel, you must have the following on your host PC:

1. vmlinux – the Linux kernel image with debug symbols.
2. Associated Kernel source files.

Please ensure that the uImage available on the PandaBoard SD card corresponds to the vmlinux file used for debugging.

5.2.1 Kernel Debug

1. In Sourcery CodeBench, create an empty project **File|New|C Project**.
2. Provide a project name, select **Project type** as **Empty Project** and select the **Toolchains** as **Sourcery CodeBench for ARM GNU/Linux**. Click **Next**.

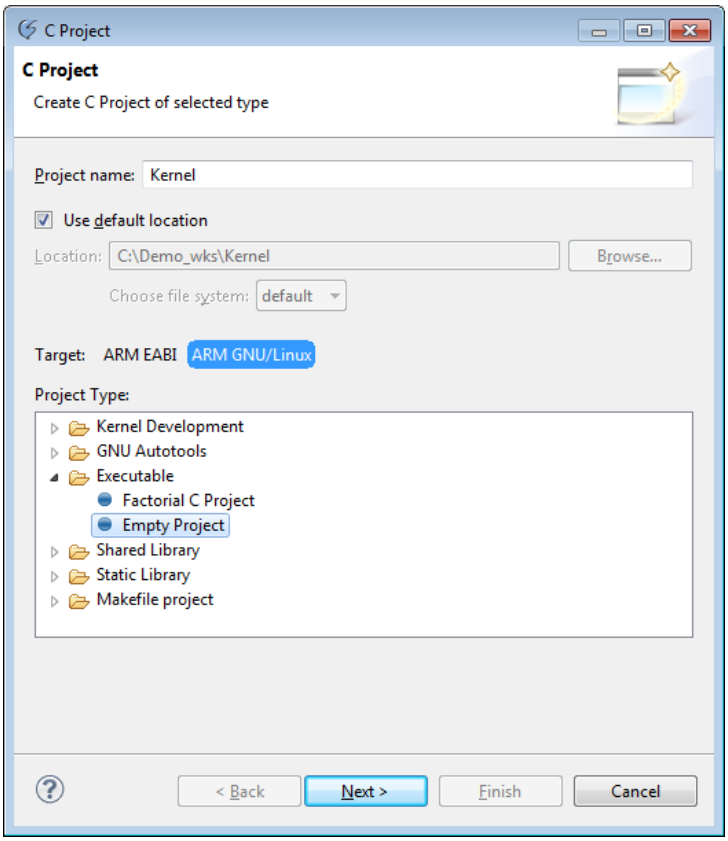


Figure 46. Kernel Project

3. Leave the defaults and click **Next**.

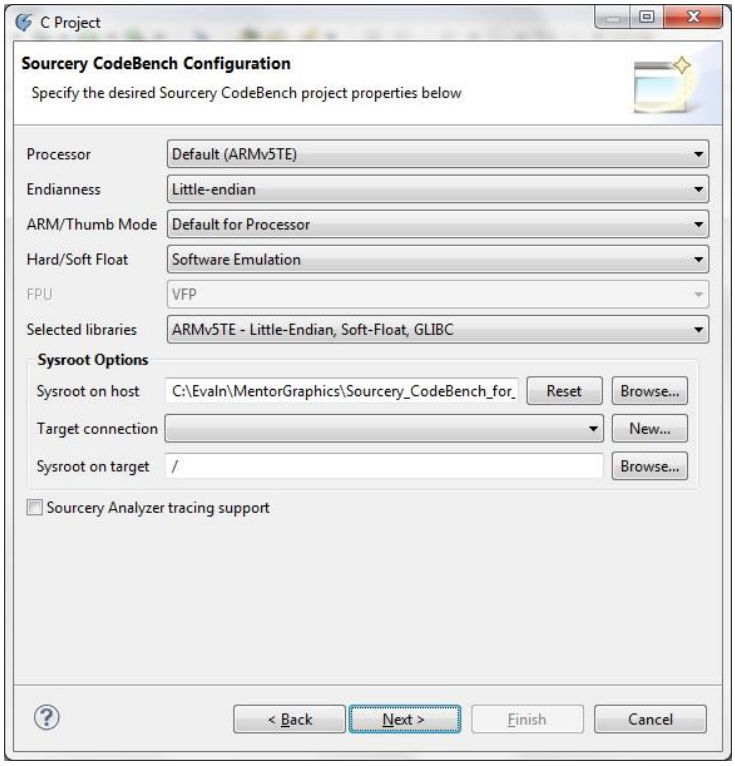


Figure 47. Kernel Project

4. Select **Launch type** as **Sourcery CodeBench Kernel Debug (Attach)**and **Debug interface** as **Ashling Vitra-XD**. Select the specific Vitra-XD to be used from the **Device** combo. Click **Finish**.

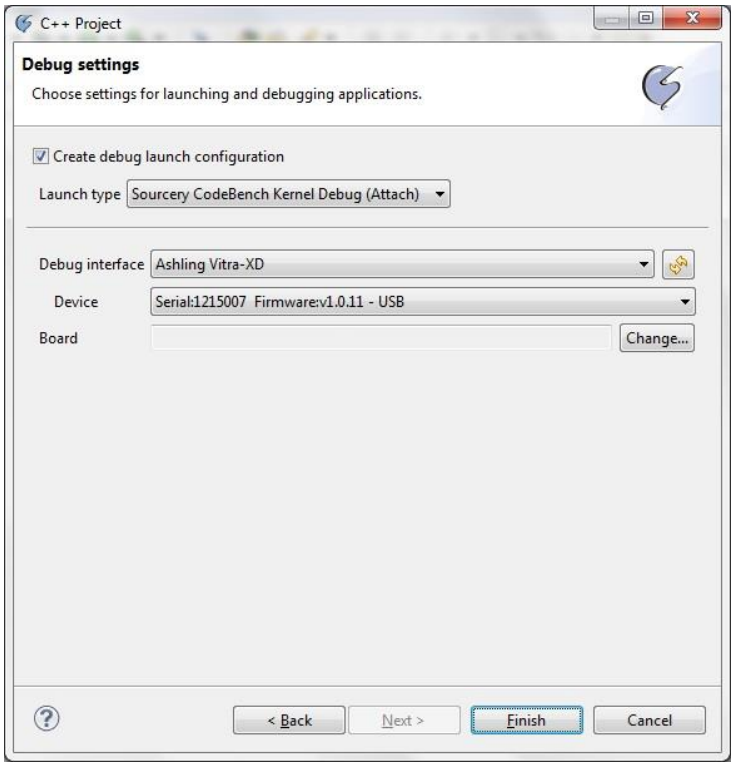


Figure 48. Kernel Project

5. From **Run|Debug Configurations**, select the **Main** tab. In **C/C++ Application,** select the vmlinux file on your host PC.

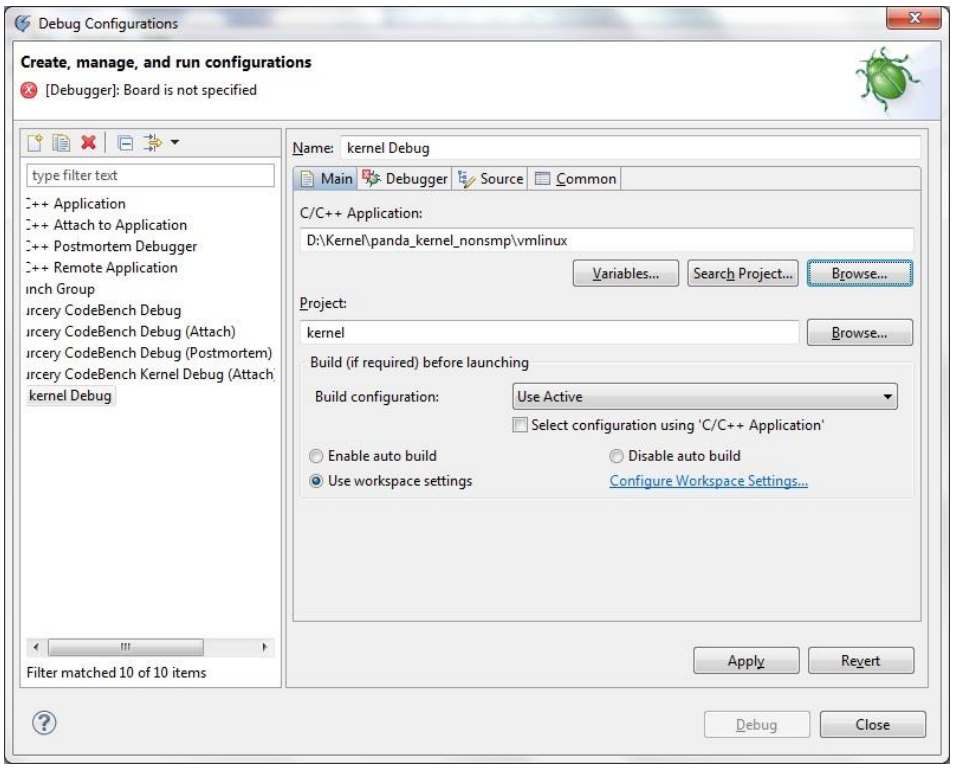


Figure 49. Debug Configuration

6. In the **Debugger** tab, select **Board** as PandaBoard using the PandaBoard.xml file available in *<Installation_Path\codebench\arm-none-eabi\lib\boards>*.
7. Select **Device** as **TI_OMAP4430_A9CORE1**. Ensure that **Reset** is set to **none** and **Semhosting** is turned **off** (in **Debugger Options|Ashling Vitra-XD)**.

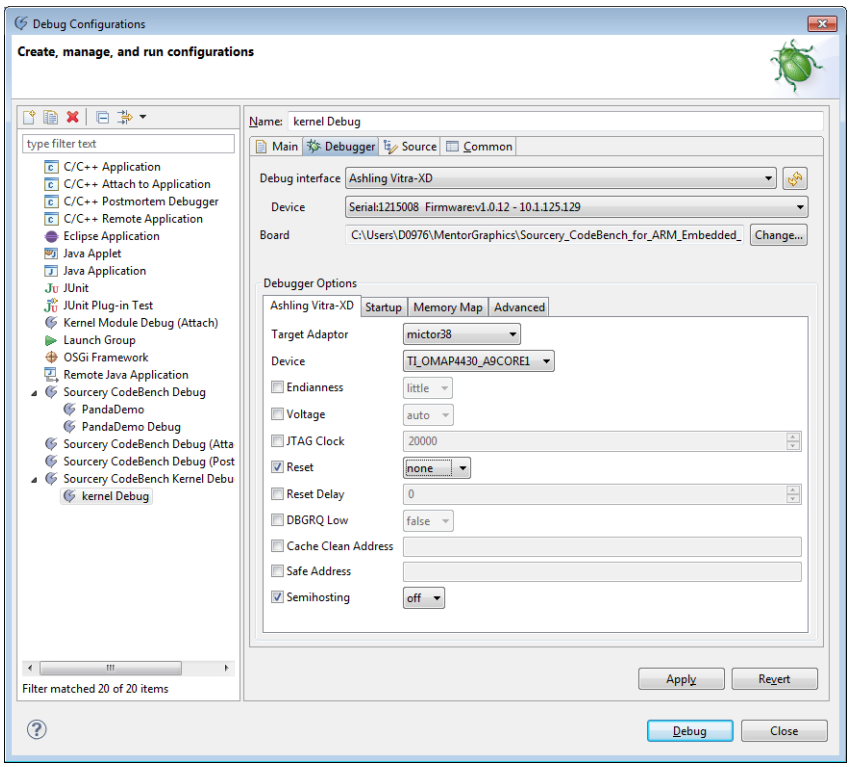


Figure 50. Debug Configuration

8. In **Source** tab, configure the path to the kernel source files. Click **Add...**.

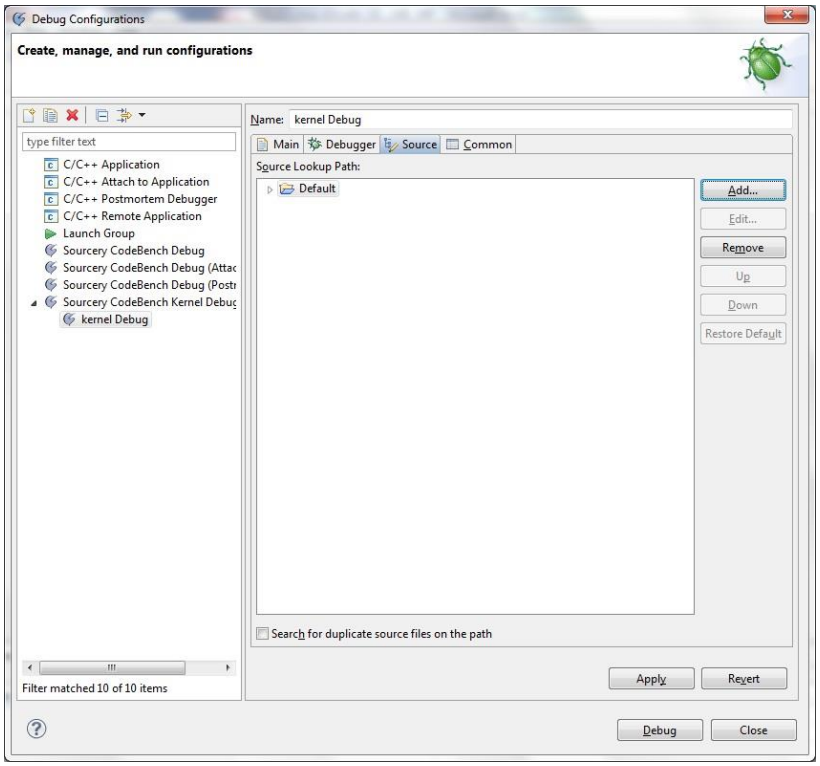


Figure 51.Source Path Mapping

9. Select **Path Mapping** and click **OK**.

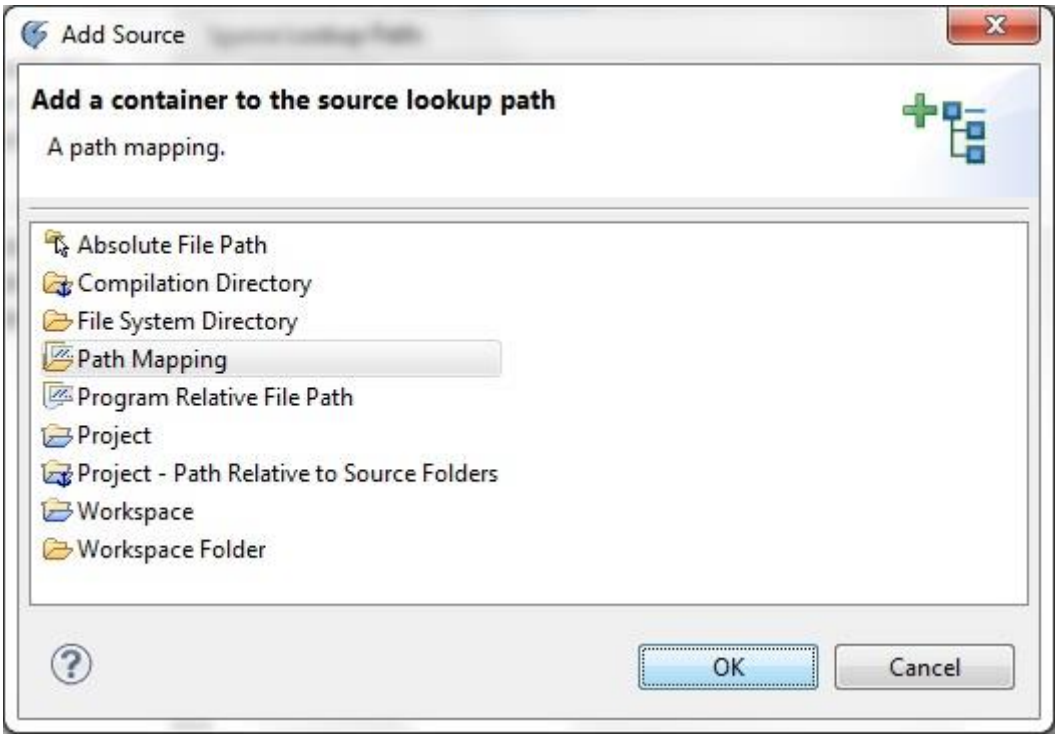


Figure 52.Source Path Mapping

10. Provide a name for path mapping configuration in **Name** and click **Add**. In **Compilation path** specify the path where the Linux kernel was compiled and in **Local file system path** specify the path to the kernel source files. Click **OK**.

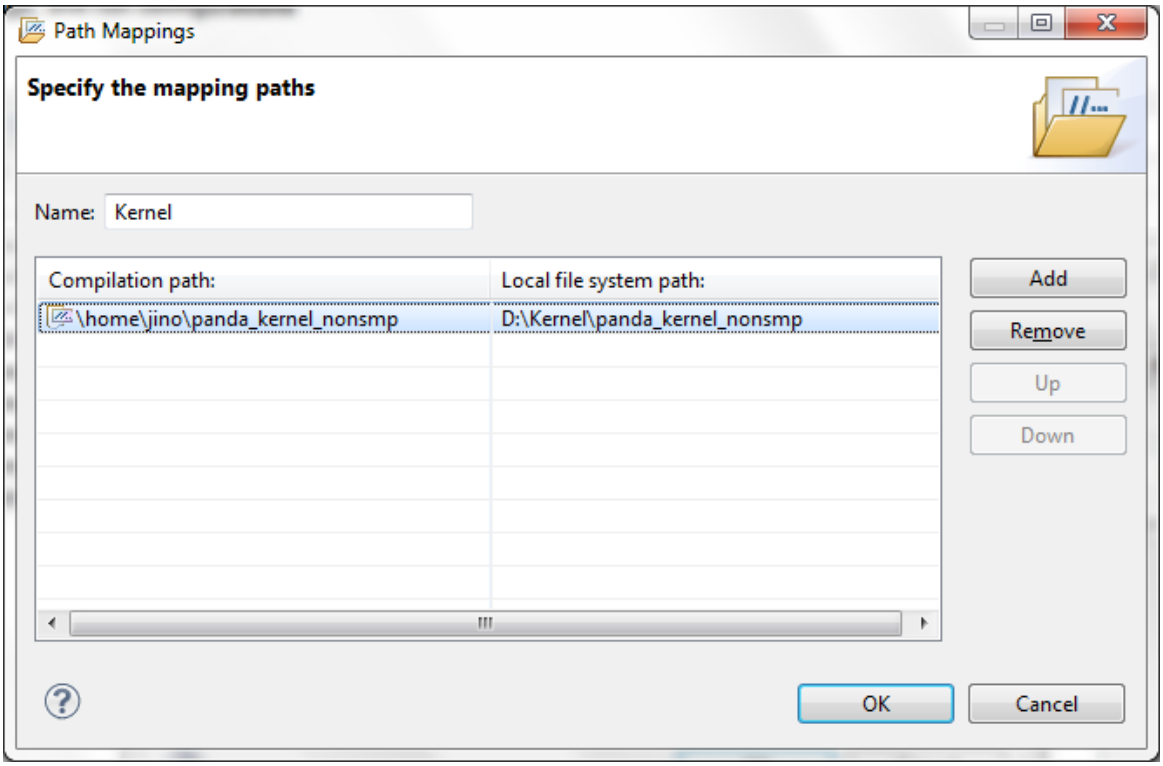


Figure 53. Source Path Mapping

11. Click **Apply** and **Debug**.

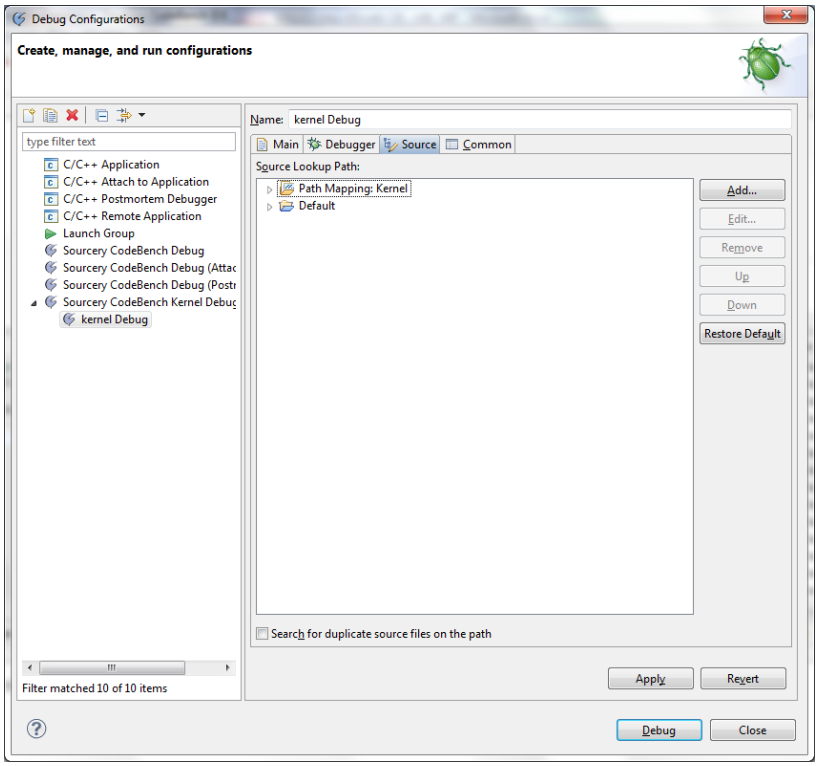


Figure 54. Source Path Mapping

12. On clicking **Debug**, Sourcery CodeBench will connect to PandaBoard via Vitra-XD and will halt the Linux kernel.

5.2.2 Kernel Trace

Kernel Trace can be configured as described in the previous sectionTrace support.

5.3 Single Process Debug and Trace

5.3.1 Create a sample project for the Linux application

1. Create a sample application using **File|New|C Project**.
2. Provide a **Project name**, select **Project type** as **Factorial C Project** and select **Toolchains** as **Sourcery CodeBench for ARM GNU/Linux**. Click **Next**.

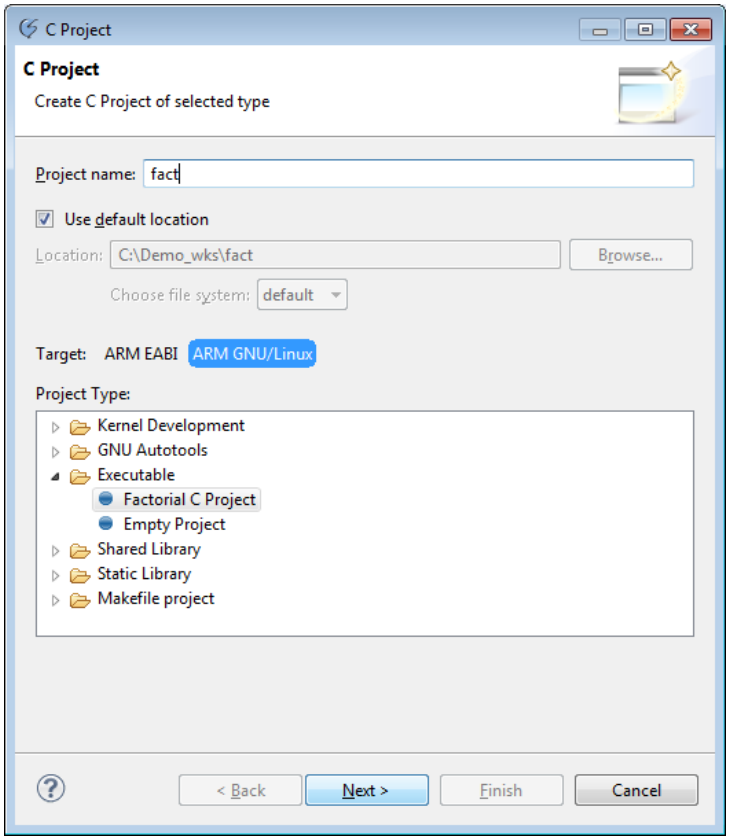


Figure 55.Application Project

3. Click on **New** in **Sysroot Options**.

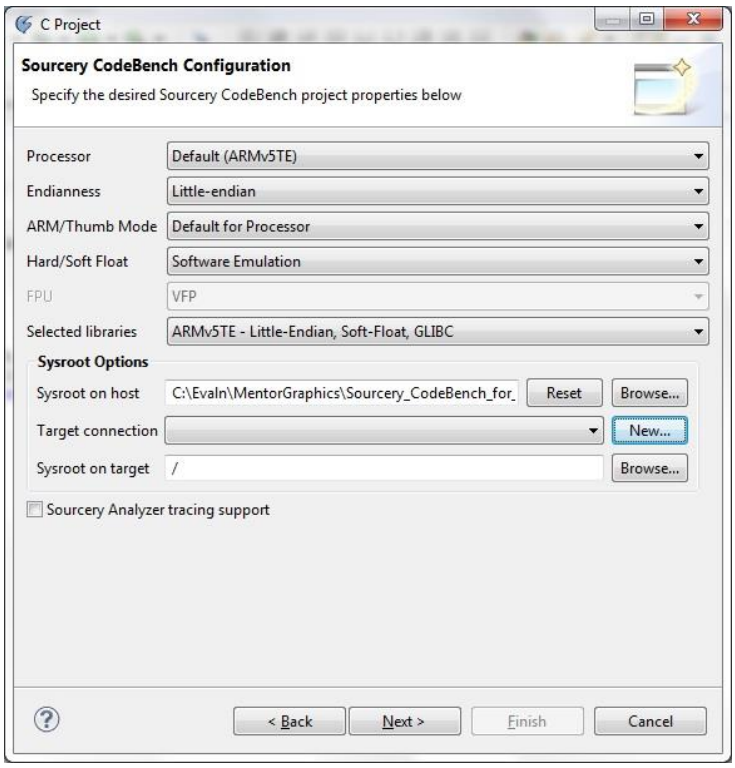


Figure 56. Configure Target connection

4. Click **Next**.

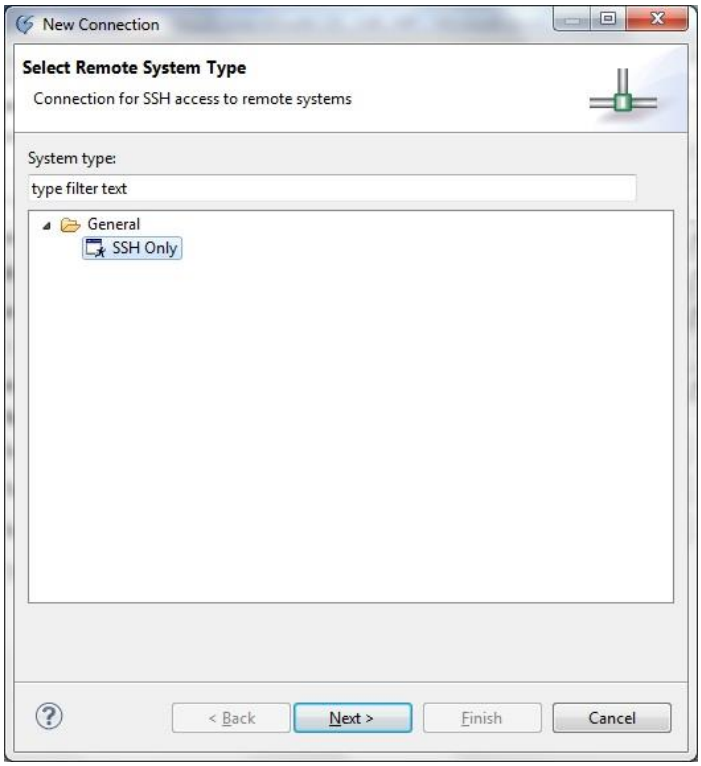


Figure 57. Configuring connection

5. Specify the IP address of the PandaBoard as **Host name** and **Connection name**. Click **Finish**.

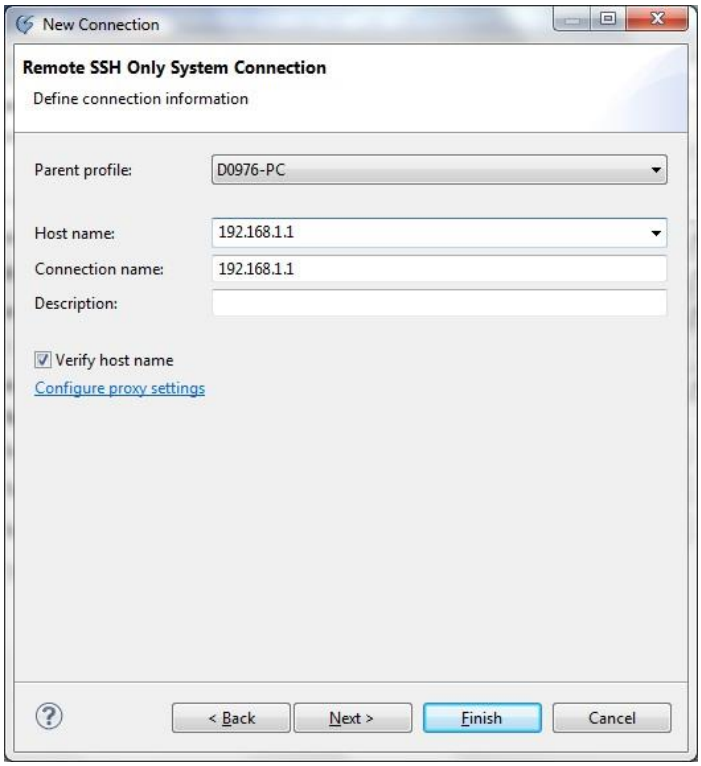


Figure 58.Sysroot configuration

6. Configure the **Sysroot on target** as /libc. You can browse the target file system by clicking on the **Browse** button.

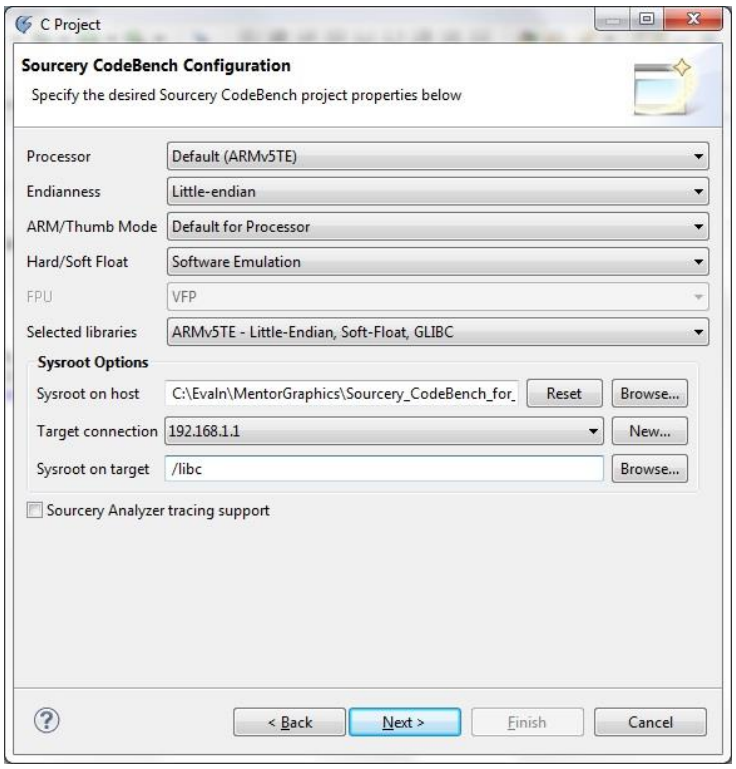


Figure 59.Sysroot configuration

For details on configuring Sysroot, please refer to the previous section **5.1.2.1Configuring the PandaBoard sysroot** for details. Click **Next**.

7. Select **Debug interface** as **Sourcery CodeBench External Server (ARM GNU/Linux)**. Configure **Host name or IP address** as your PandaBoard's IP address. Click **Finish**.

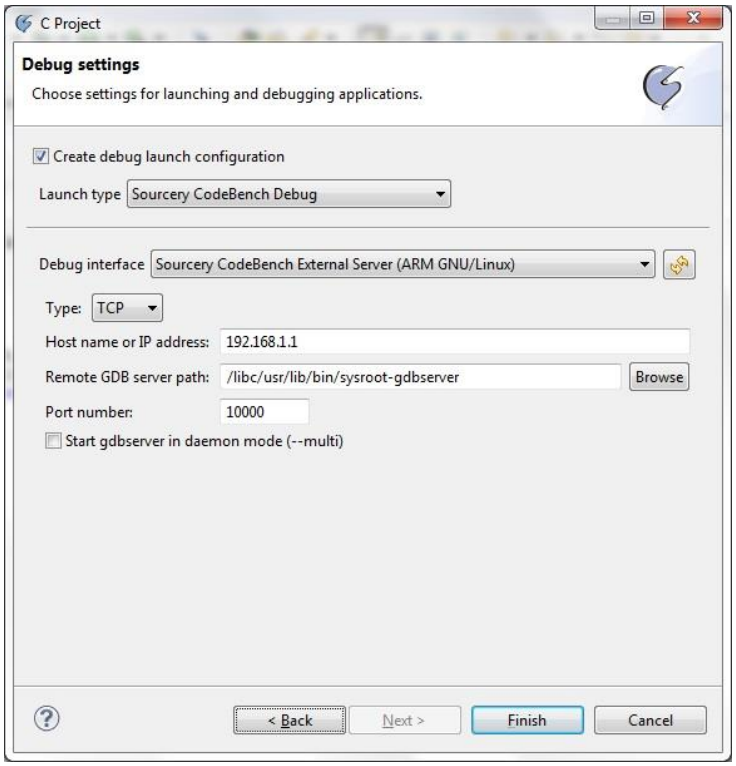


Figure 60. Debugger Configuration

8. Compile the sample application via **Project|Build Project**.

5.3.2 Debug a Linux application

1. Select **Run|Debug Configurations**. Click **Debug**.

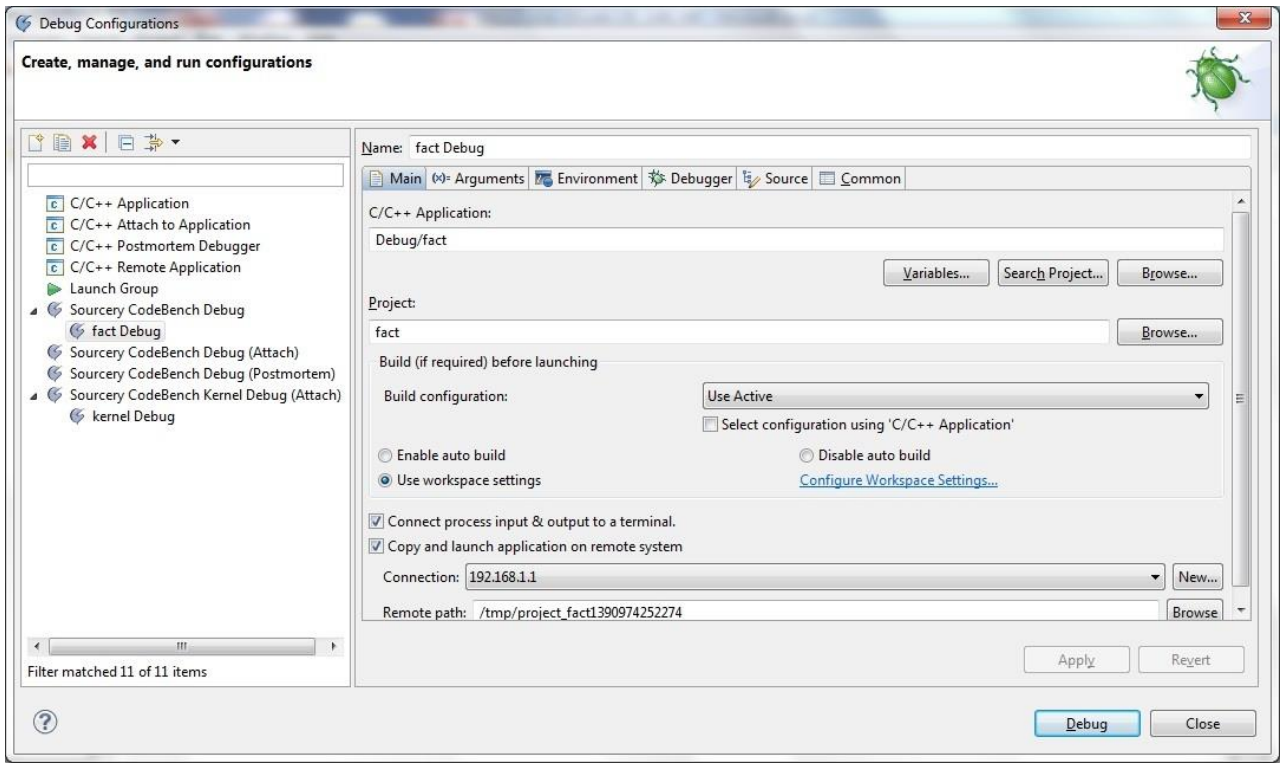


Figure 61. Debug Configuration

When prompted for a password, set **User ID** as `root` without any password. Click **OK**.

Figure 62. Login

2. Certify the authenticity by clicking **Yes** to proceed.

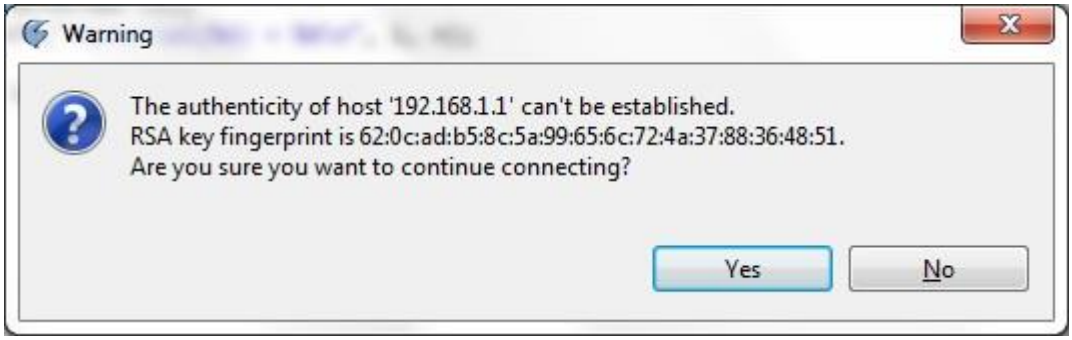


Figure 63. SSH certificate

3. The Process will be halted at the **main()** function of the application allowing debug.

5.3.3 Tracing a Linux Process

To enable debugging and tracing of a process ensure you configure the **Sysroot Options** correctly. Refer to the previous section **5.1.2.1Configuring the PandaBoard sysroot** for details.

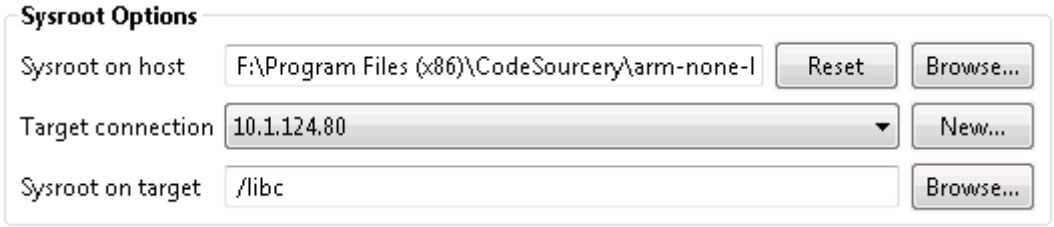


Figure 64.Sysroot options

During an active process debug session, you can choose to trace a specific Linux Process in the **Debug** view by right clicking over the process and selecting **Start Trace** as shown below:

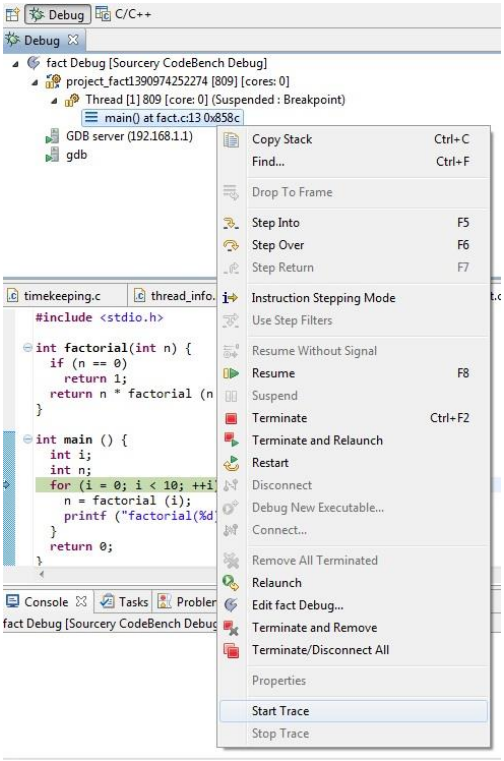


Figure 65. Start a process trace session

This will bring up the **Advanced Trace Configuration** dialog where you will be able to configure Vitra-XD for capturing trace; this dialog will not be invoked if you have previously configured a trace session. Once started, tracing may be stopped by right-clicking over the process and selecting **Stop Trace.**
Please note:

1. SMP Linux is currently not supported.
2. Process tracing includes tracing of any used shared libraries.
3. A kernel debug session with Vitra-XD must be configured prior to process tracing. If you attempt to **Start Trace** without doing this, you will be automatically taken to the **Kernel launch configuration**.

5.4 Shared Library Debug and Trace

5.4.1 Debugging a shared library

1. Create a shared library via **File|New|C Project.**
2. **P**rovide a **Project name**, select **Project type** as **Shared Library|Empty Project and select Toolchains** as **Sourcery CodeBench for ARM GNU/Linux**. Click **Finish.**

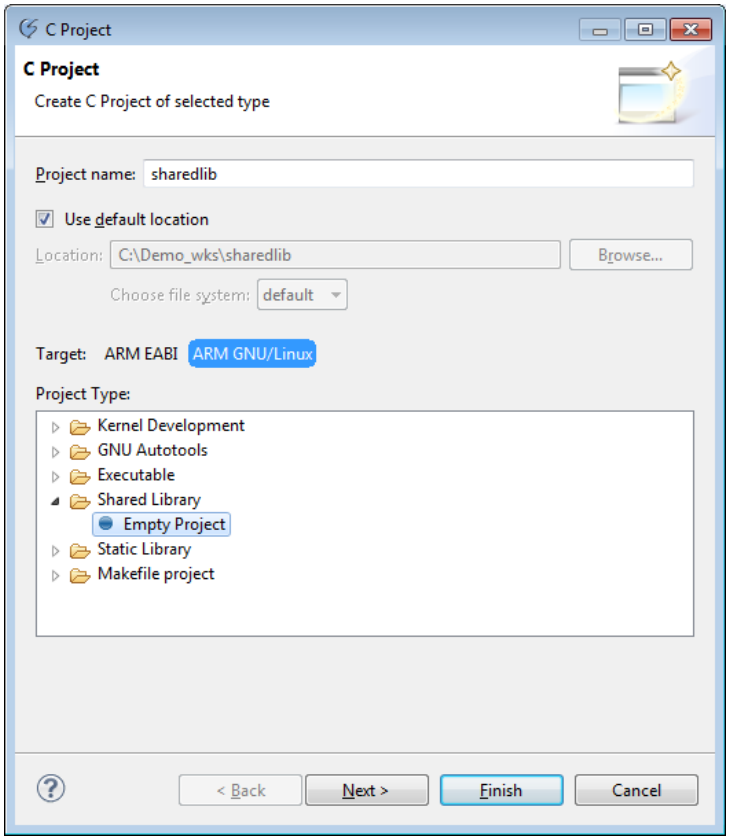


Figure 66. Shared library project

3. Right click on the project in **Project Explorer** and add C file via **New|File**.

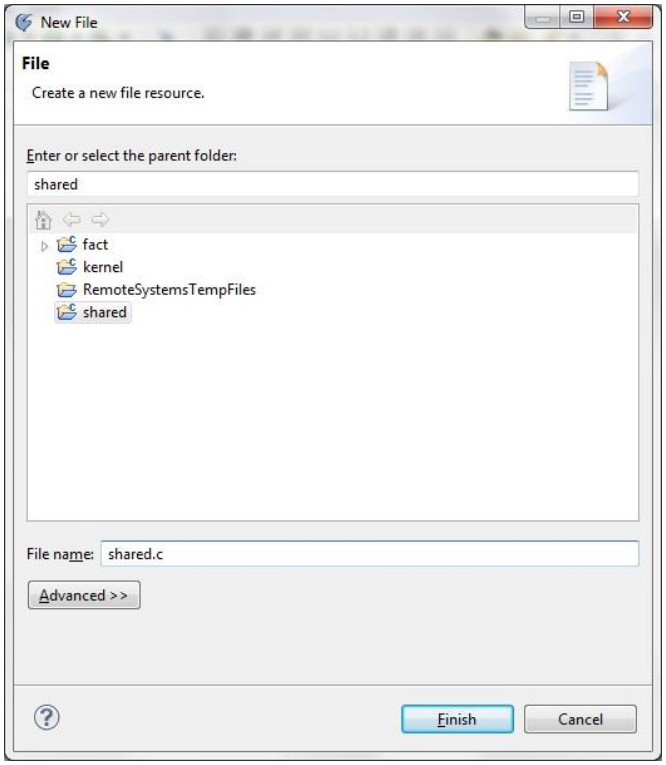


Figure 67.Adding file

4. Populate the file with the relevant code.

5. Compile the shared library via **Project|Build Project**
6. Copy the shared library to the target file system (*/lib*) using FTP or SSH.
7. To link the shared library to your application, right click on the application project in the **Project Explorer** and select **Properties**.
    a. In **C/C++ Build|Settings|ToolSettings|Sourcery CodeBench C Linker|Libraries,** set **Libraries (-l)** as library name (without the prefix *lib* and extension *.so*) and **Library search path (-L)**as the **Debug** folder in the workspace for shared library. Click **OK**.

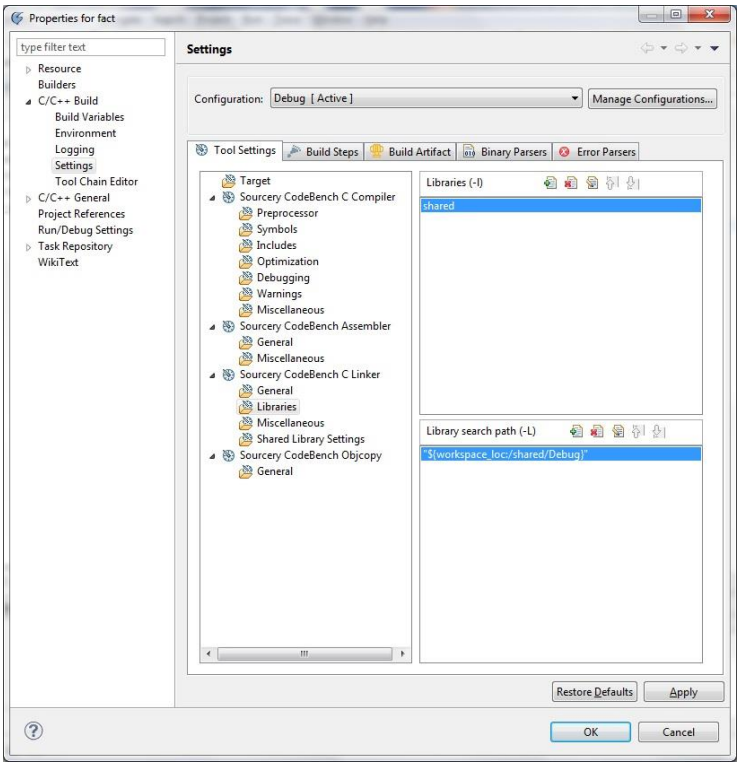


Figure 68. Configuring shared library in application

b. Enable your shared library project in the **Project References**. This will ensure that the shared library will be automatically compiled whenever the application is compiled.

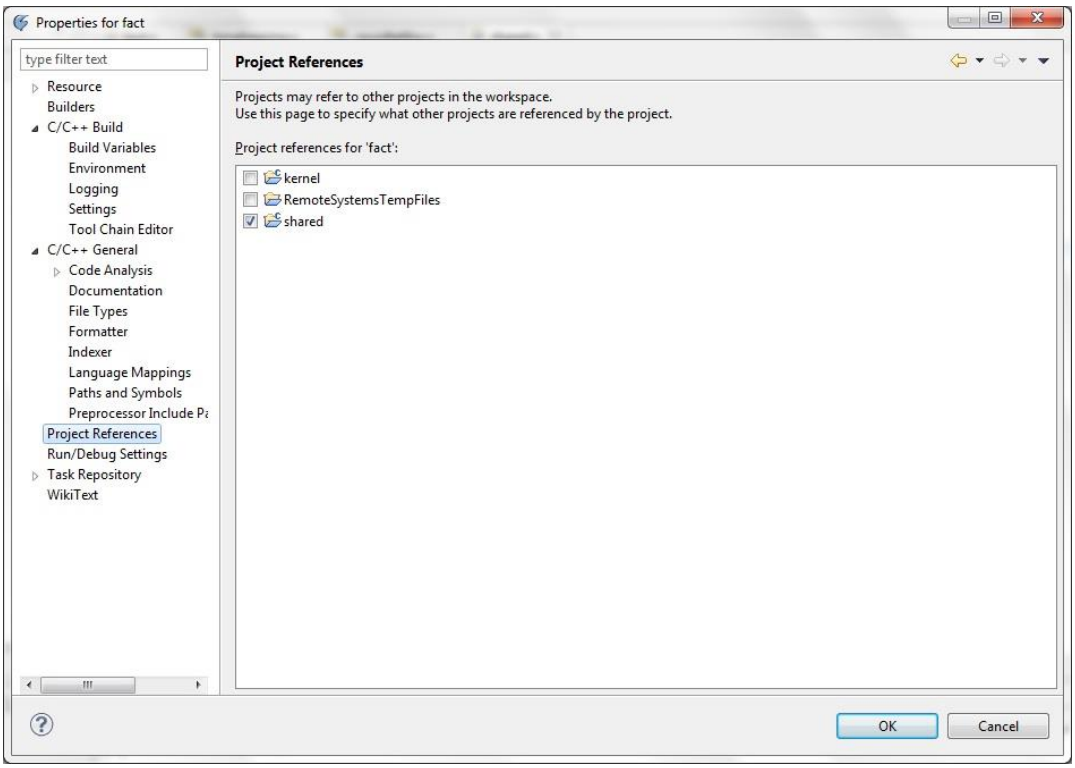


Figure 69. Project References

c. Enable your shared library project in the **Paths and Symbols** of the shared library.

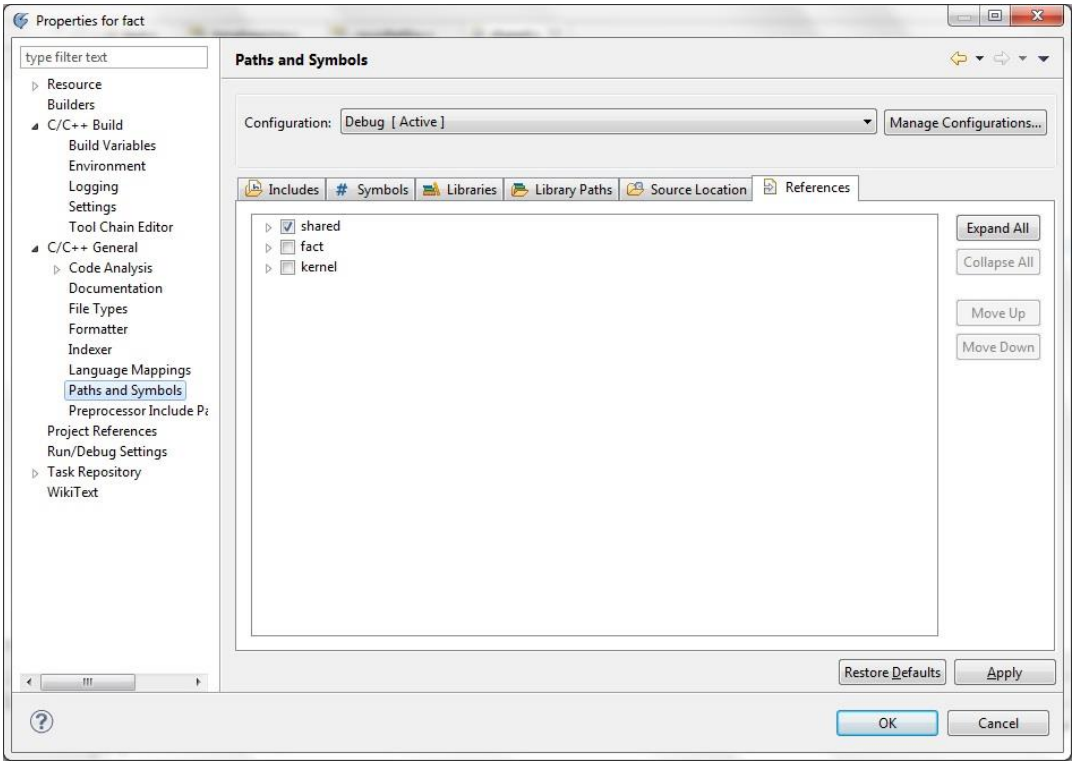


Figure 70. Paths and Symbols

d. Click **OK**.

8. To debug the shared library, its path has to be added to the **Debug Configurations|DebuggerOptions|SharedLibraries|Additional shared library directories**.

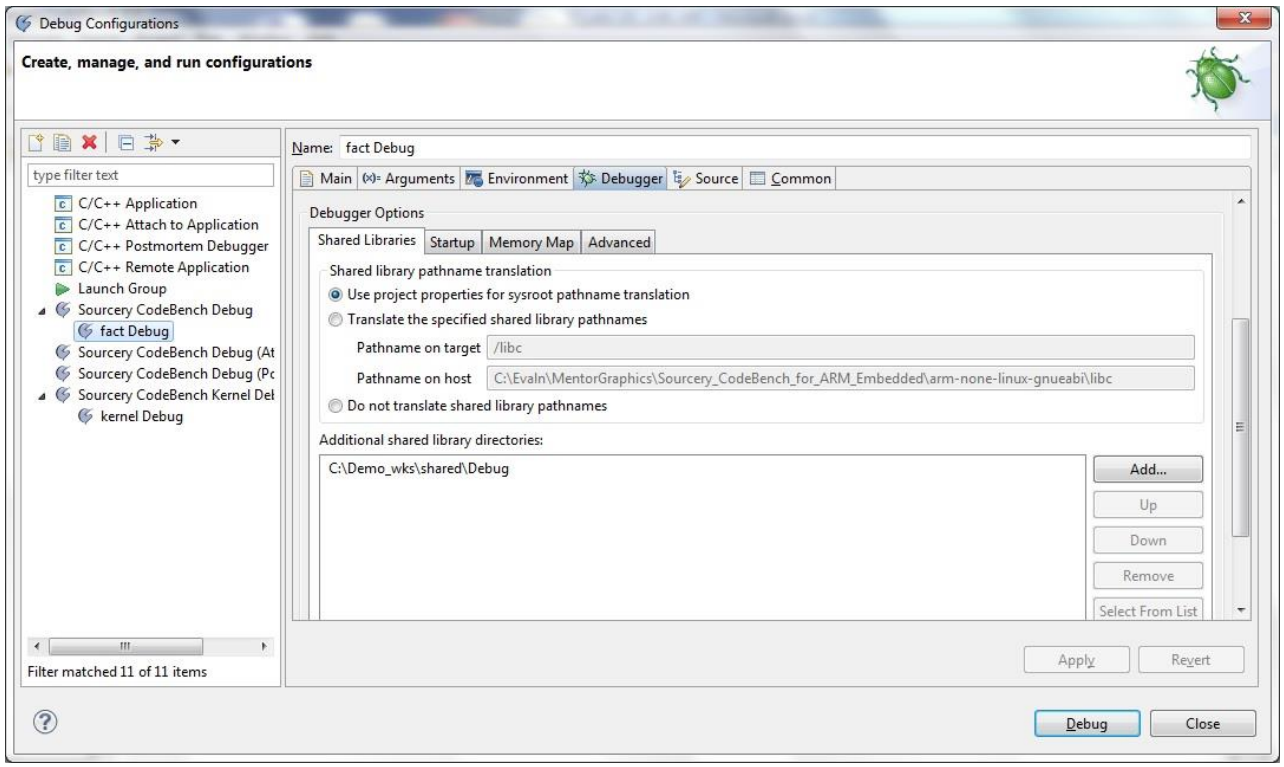


Figure 71. Debug Configurations

9. Build the application which the shared library is linked to and debug it as mentioned in the previous section **5.3.2Debug a Linux application.**

5.4.2 Tracing a shared library

Trace the application (process) which the shared library is linked to as mentioned in section **5.3.3Tracing a Linux Process**. In the trace view, trace from shared library will be marked by its name and the PID of the process which loaded it.

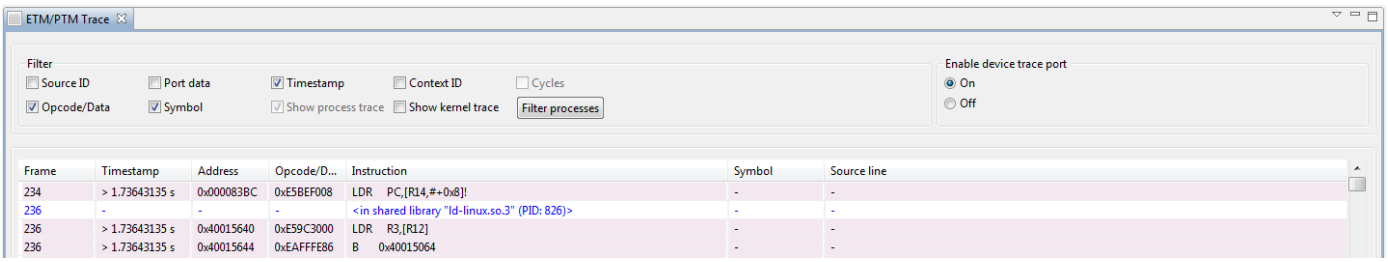


Figure 72. Shared library in trace view

5.5 Simultaneous Kernel and Process Debug and Trace

To debug both kernel and process, you must launch two debug configurations – one for debugging kernel and another for debugging process. Configuring kernel and process launch are explained in the previous sections.

To trace both kernel and process, you have to configure single process tracing, as mentioned in the section **5.3.3Tracing a Linux Process**. Captured trace will have instructions from both kernel and process. In the trace view, you can view the trace of process alone, kernel alone or both process and kernel using the filters provided.

Figure 73. Kernel/Process filters

5.6 SMP Kernel Debug

The steps for configuring and running an SMP Kernel debug session are the same as that in Non-SMP Kernels, except that the user has to configure an SMP Kernel image and specify the cores in the Debug configuration. Please refer to section **5.2.1** for a description of Kernel debugging.

5.7 SMP Process Tracing

Please refer to the previous section **5.3.3Tracing a Linux Process.** The ETM/PTM Trace view will show the associated **Source ID** for a traced process. Cores are automatically assigned a Source IDs of the core + 1.

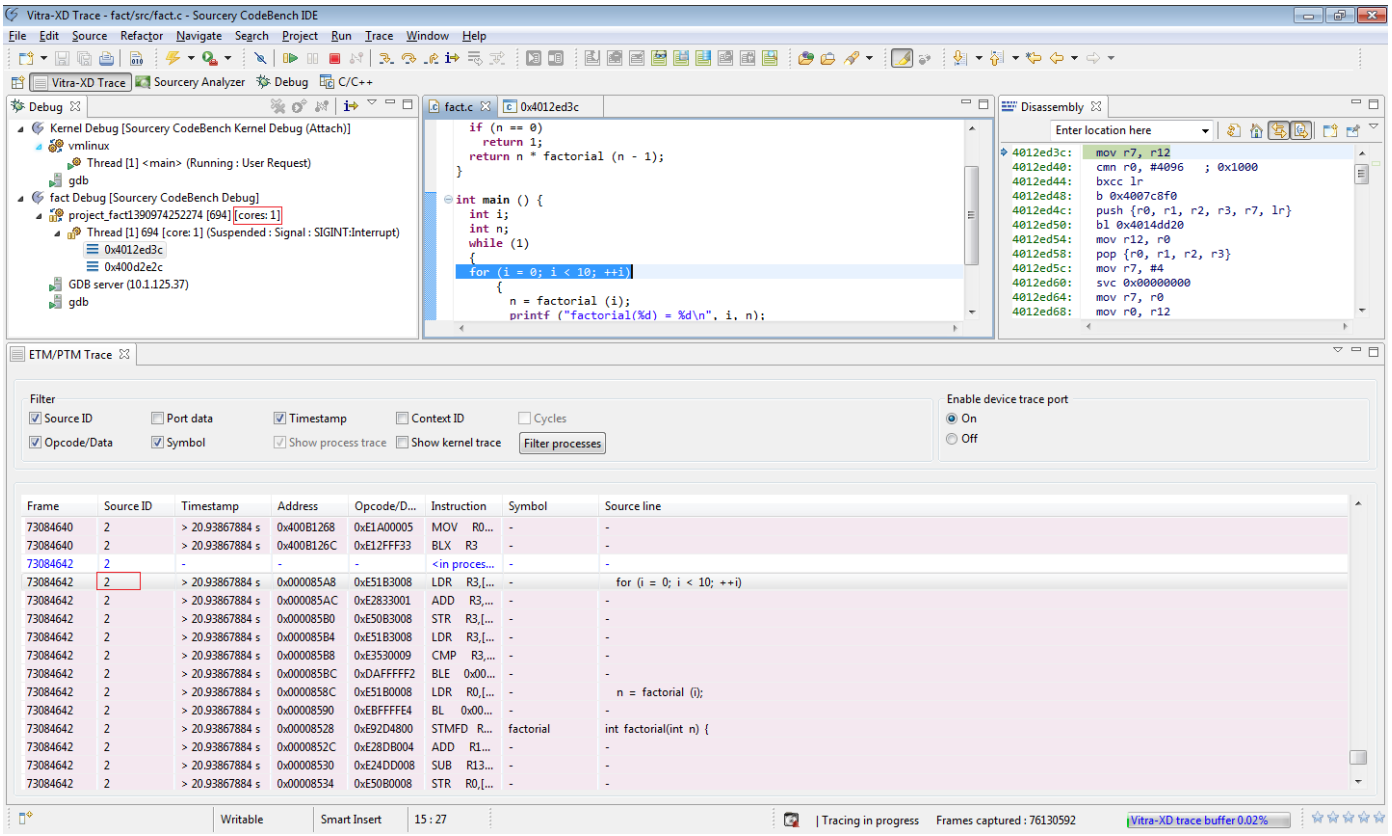


Figure 74. SMP process tracing

6. Multi-core Debugging

This section describes how to use Vitra-XD with Sourcery CodeBench to debug a multi-core device. The example given is based on a TI OMAP4460 based PandaBoard target.

The initial steps like connecting to the target, setting up the CodeBench workspace and building the project are the same as in a normal debug session. For details, please refer sections **3.1** and **3.2** . For multi-core debugging, you need to specify the cores to be debugged in the debug configuration dialog as shown below:

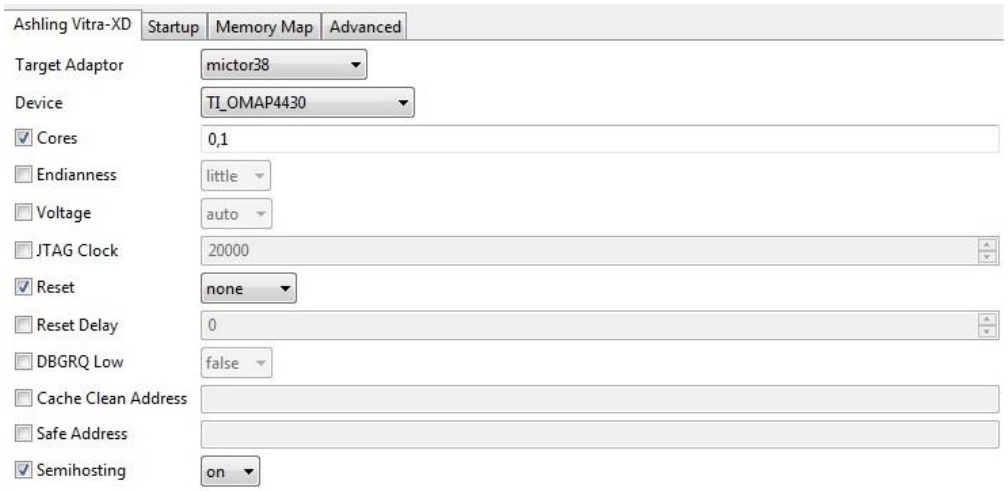


Figure 75. Debugger options for multi-core debugging

Once the debug session is started, the cores will be listed as separate trees in the debug view.

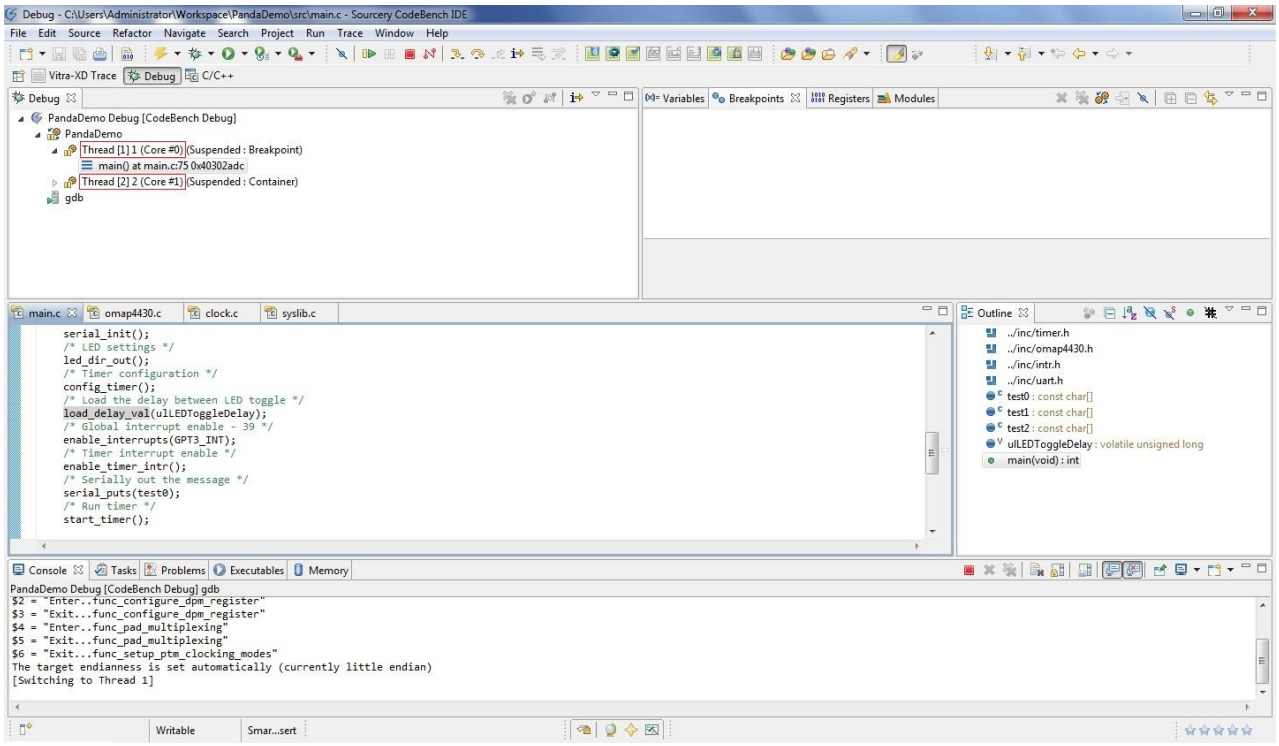


Figure 76. Debug perspective for multi-core debugging

You can switch between different threads just by selecting them and do run-time debug operations like *step-in*, *step-over* etc. on specific cores.

7. Using Vitra-XD with the Xilinx ZC702 board

7.1 Introduction

This section describes using the Xilinx ZC702 board and Vitra-XD to debug and trace software running on the Zynq ARM core.

7.2 Hardware setup

7.2.1 Setting up ZC702 board

1. Vitra-XD requires the FMC XM105 debug card (Xilinx HW-FMC-XM105-G) to be connected to the *J3 FMC1* connector on ZC702 board and screwed on to the board firmly. The FMC board provides the mictor connector needed for tracing.

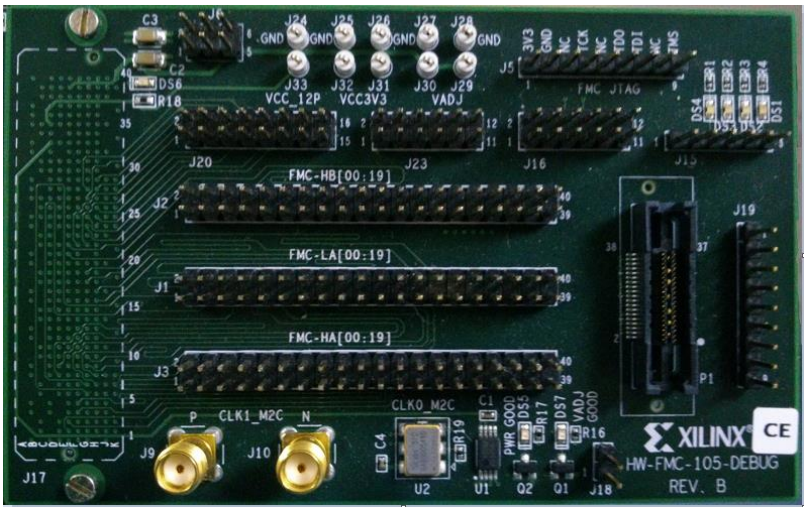


Figure 77: FMC XM105 debug card

Note: An additional adapter board (provided by Ashling) is needed to bring the debug (JTAG) lines to the mictor connector. This adapter:

- Connects pin-headers J16 and J19
- Connects TDI to TDO pins on pin-header J5

The below figure shows the adapter and FMC debug card plugged into the ZC702 board:

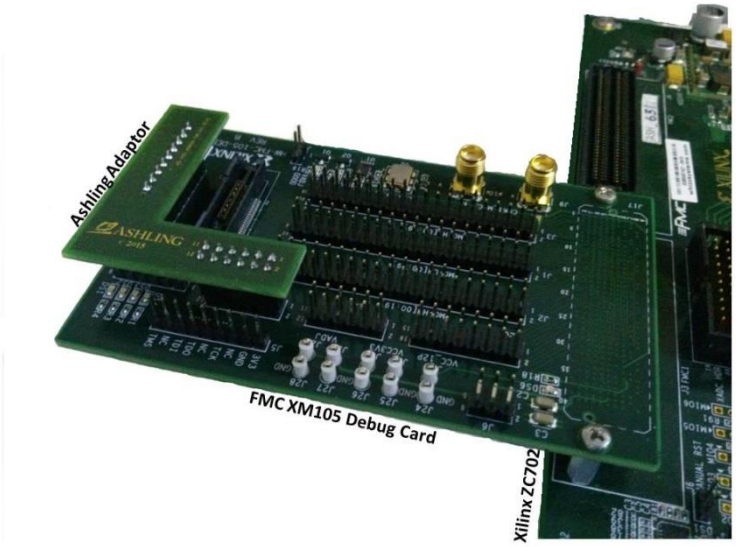


Figure 78: FMC debug card plugged on Xilinx ZC702 board

7.2.2 Programming the Zynq FPGA

1. The Zynq FPGA must be programmed with an appropriate core which routes the trace signals to the mictor connector. Ashling provide a suitable FPGA file to do this.
    a. Copy the Zynq FPGA file *BOOT.BIN* from the folder *<Installation_path>\codebench\i686-mingw32\arm-none-eabi\ash\demos\xilinx_zc702\fpga* to an empty SD card.
    b. Insert the SD card in to the *SD card slot* (J64) on ZC702 board.
    c. Change the boot settings (*SW16*) on the board to boot from SD card as follows:

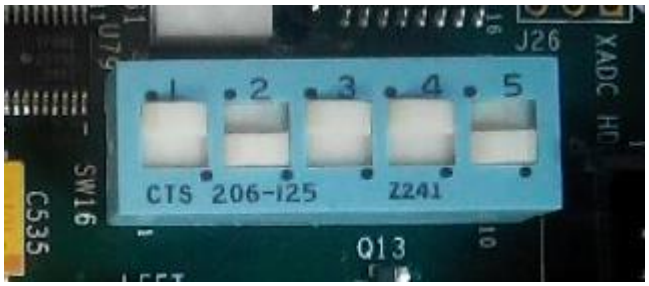


Figure 79: Boot mode configuration

7.2.3 Connecting the Vitra-XD to ZC702 board

The Vitra-XD is connected to the mictor connector on FMC debug board as shown below:

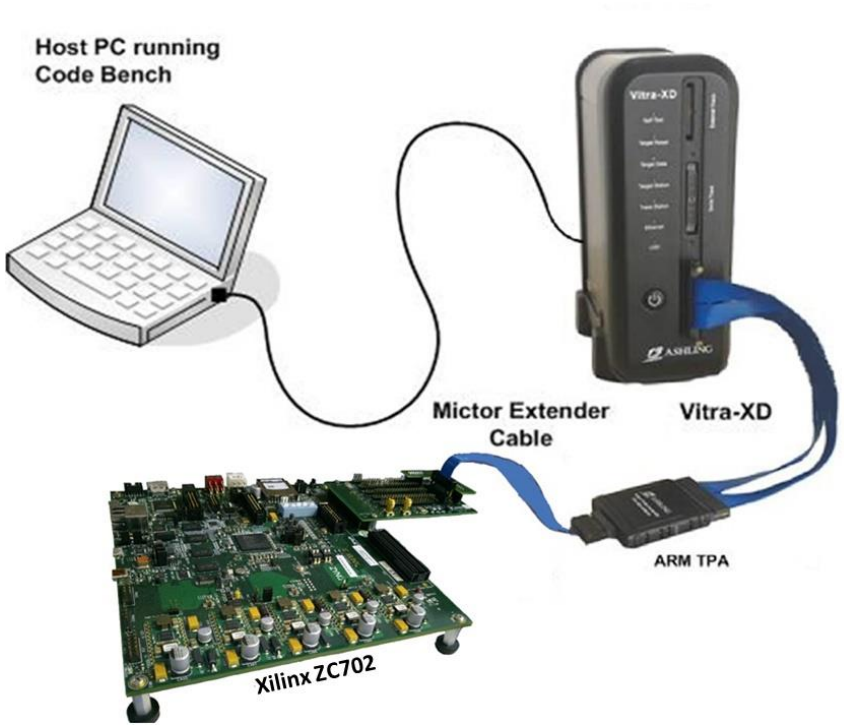


Figure 80: Vitra-XD/ZC702 hardware setup

7.3 Software setup

7.3.1 Debugging

Ashling provide two suitable demo projects for the Xilinx ZC702 board in `<Installation_path>\codebench\i686-mingw32\arm-none-eabi\ash\demos\xilinx_zc702\demos`. These include a single-core demo (`ZC702DEMO`) and a multi-core demo (`ZC702DEMO_MC`).

To import, build and debug the required project please refer to Section 3.2.1.*Note*: Choose the **XILINX_ZYNQ7000Device** in the ***Debugger|Ashling Vitra-XD*** as shown below when debugging.

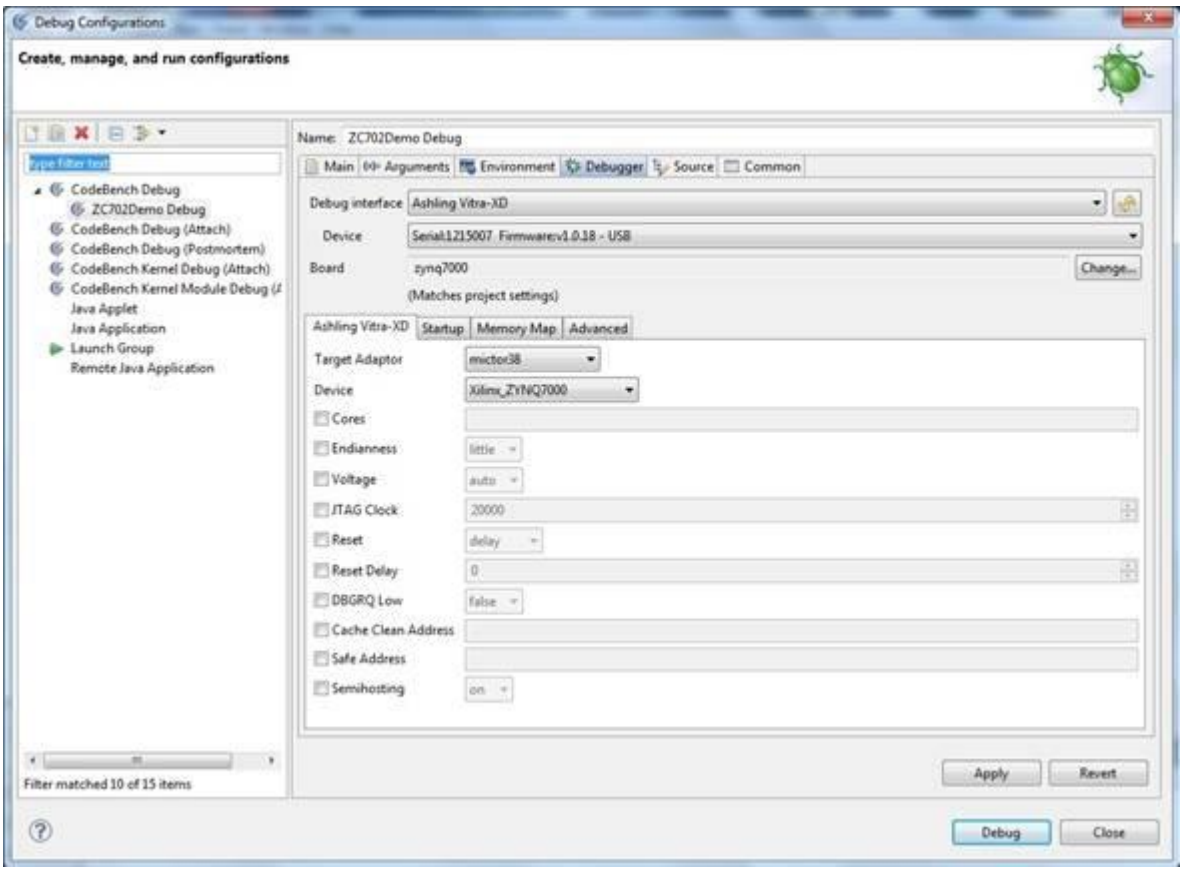


Figure 81: Debug Configuration (single-core)

For multi-core debugging, specify cores 0 and 1 as follows in **Cores:**

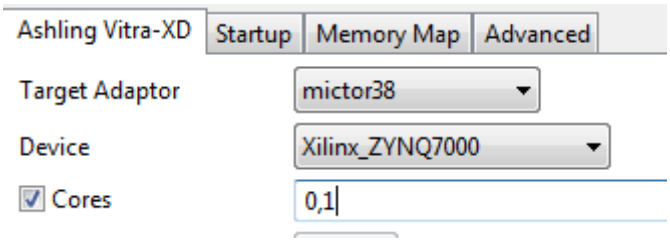


Figure 82: Debug Configuration (multi-core)

7.3.2 Tracing

Trace can be quickly configured via **Trace|QuickTrace|Trace upto last instruction**. This captures (traces) up to the last executed instruction. For advanced trace configuration options, please refer to Section 4.*Note*: Trace pins are enabled by default on the Xilinx ZC702 board, hence, no additional GDB script is needed.

8. Using Vitra-XD with the Freescale Vybrid board

8.1 Introduction

This section describes using the Freescale Vybrid (TWR-VF65GS10) board and Vitra-XD to debug and trace software running on either ARM Cortex-A5 core or Cortex-M4 core.

8.2 Connecting Vitra-XD to Freescale Vybrid

The Vitra-XD is connected to the mictor connector on Freescale Vybrid board as shown below:

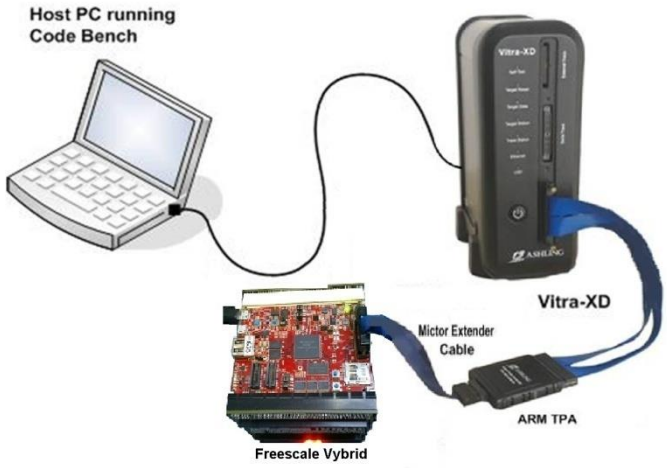


Figure 83: Vitra-XD/Freescale Vybrid hardware setup

8.3 Software setup

8.3.1 Debugging

Ashling provide suitable demo projects for the Cortex-A5 and Cortex-M4 cores in Freescale Vybrid board in `<Installation_path>\codebench\i686-mingw32\arm-none-eabi\ash\demos\freescale_vybrid`.

To import, build and debug the required project, please refer to Section 3.2.1.

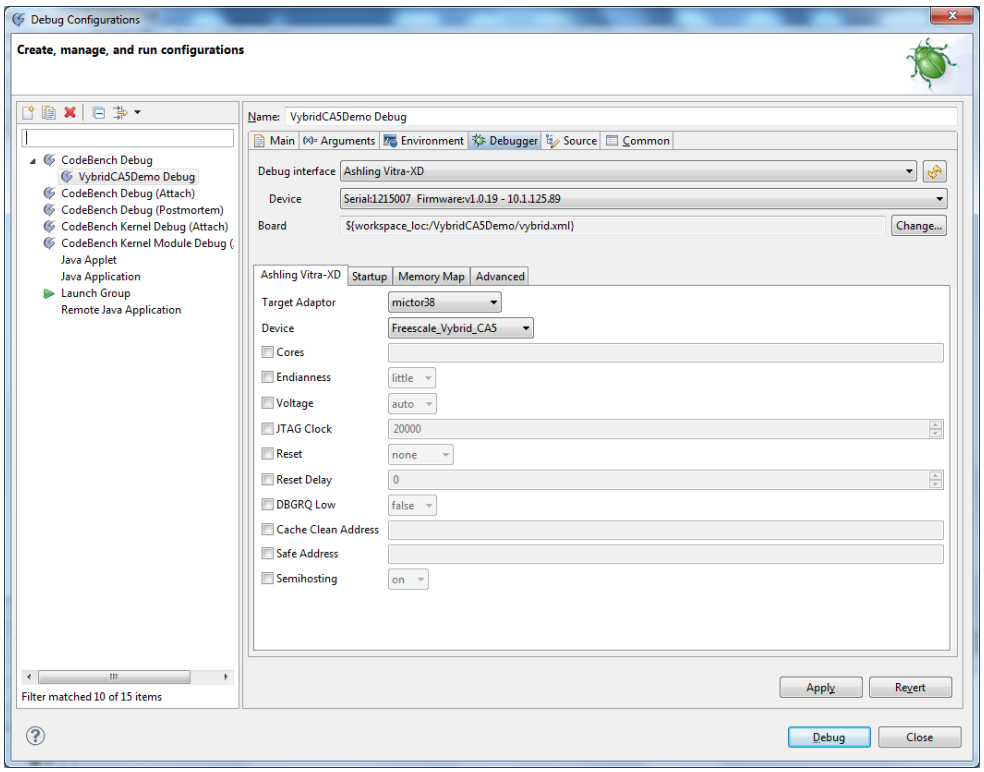


Figure 84: Debug Configuration

Note: In *Debug Configurations*, specify the board specific configuration file in **Debugger|Board** by clicking on the **Change** button.

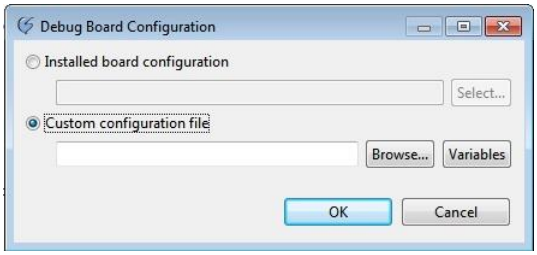


Figure 85: Debug Board Configuration

Based on the core to be debugged, select the board configuration file through **Custom configuration file|Browse**. The board configuration file for Cortex-A5 will be available in `<workspace>\VybridCA5Demo\vybrid.xml` and for the Cortex-M4 in `<workspace>\VybridCM4Demo\vybrid.xml`.

8.3.2 Tracing

Trace can be quickly configured via **Trace|QuickTrace|Trace up to last instruction**. This captures (traces) up to the last executed instruction. For advanced trace configuration options, please refer to Section 4. *Note*: Trace pins are not enabled by default in Freescale Vybrid board. To enable trace pins, run the GDB script provided in `<Installation_path>\codebench\i686-mingw32\arm-none-eabi\ash\gdbscripts\vybrid` using **GDB cmd** as shown below (don't forget to prefix with *source)*:

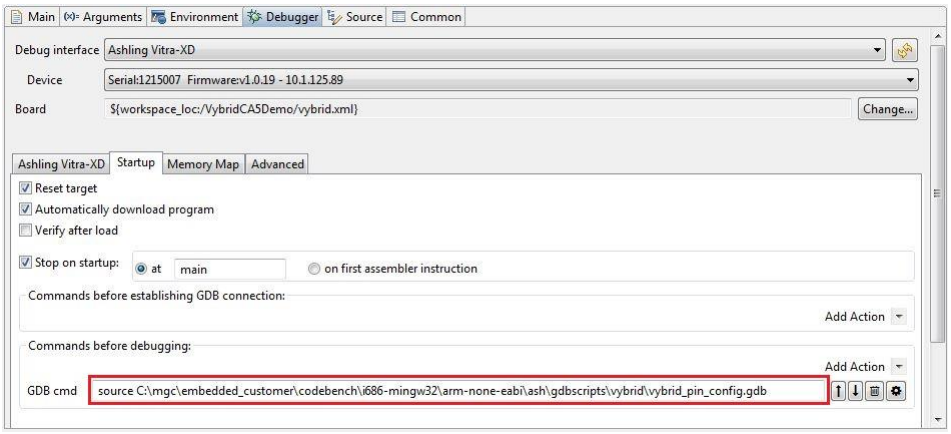


Figure 86: Enable target trace pins

8.3.3 Creating a new project for the Vybrid board

This section describes step by step procedures to create a new Cortex-A5 based project for a Freescale Vybrid board.

Note: For creating a new Cortex-M4 based project then follow the same steps but use Cortex-M4 as the generic processor in step 5 below instead of Cortex-A5.

1. Right click on **Project Explorer** and select **New|C Project.**

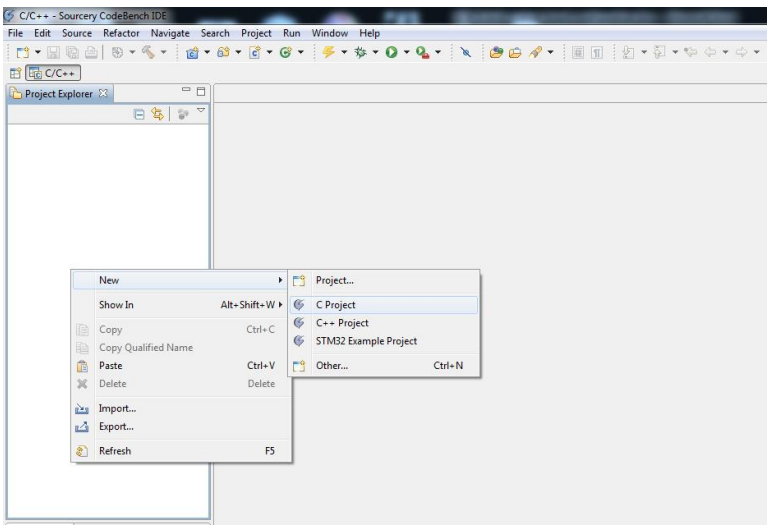


Figure 87: Create a new project

2. Select **Target** as **ARM EABI** and select **Project Type|Executable|Factorial C Project**. Click **Next**.

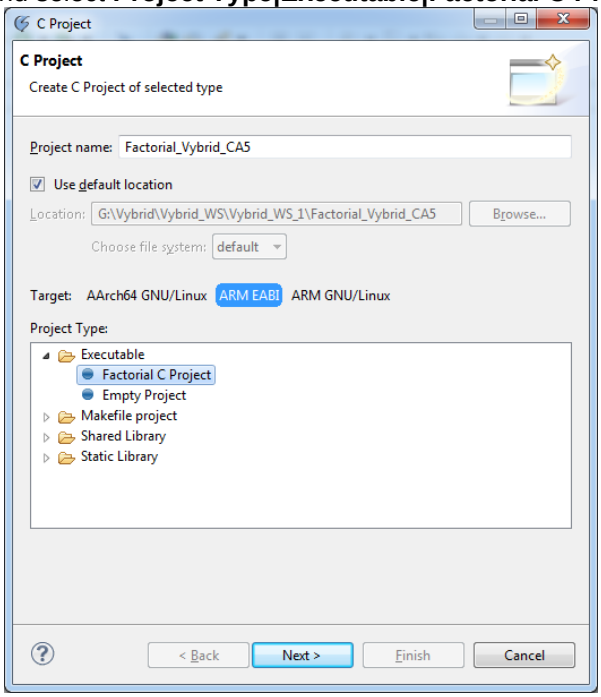


Figure 88: C Project

3. Click on **Board...**.

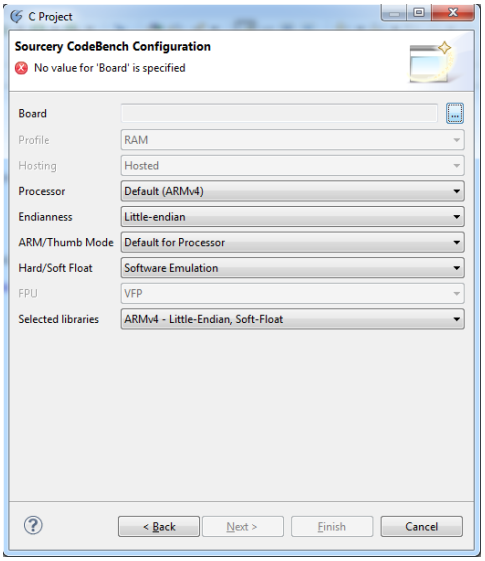


Figure 89: Board Configuration

4. Click on **Create Board**.

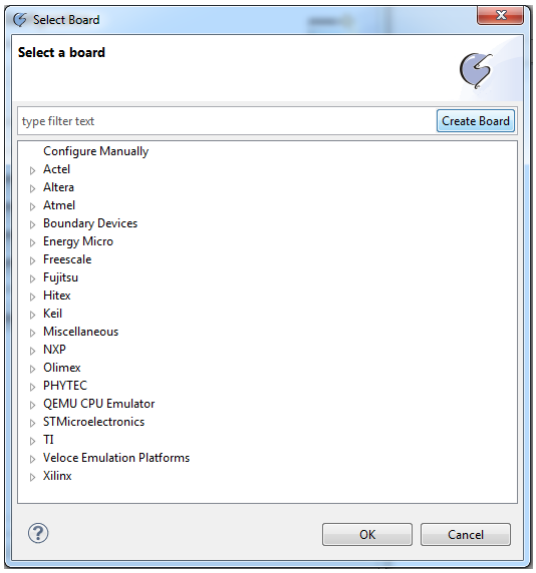


Figure 90: Create Board

5. Specify a **Board Name**(e.g. *Freescale_Vybrid_CA5*) and select **Cortex-A5** as **Base New Board on|Generic Board with Processor**. Click **Finish**.

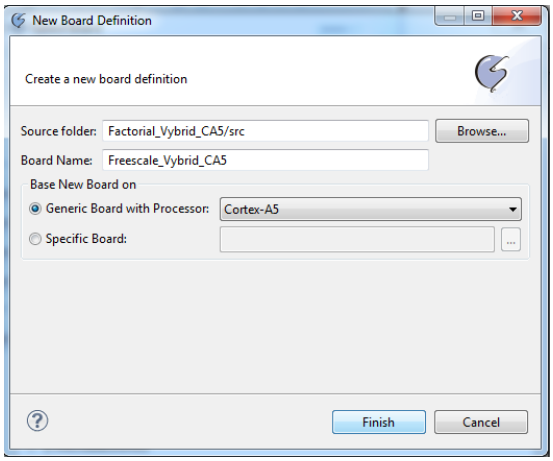


Figure 91: New Board Definition

6. Select the newly created board and click **OK**.

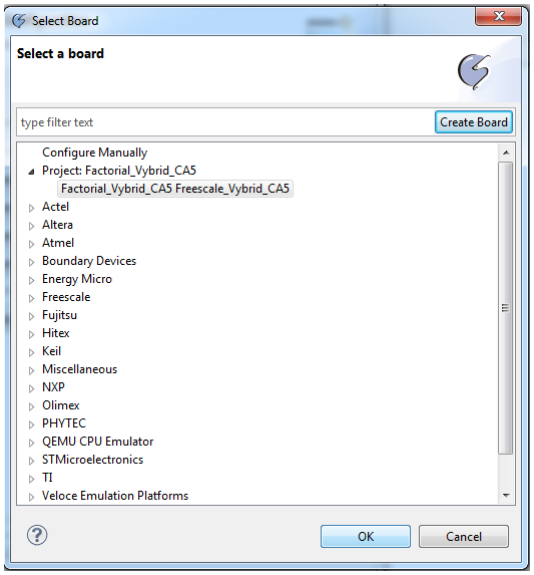


Figure 92: Select Board

7. Click **Next.**

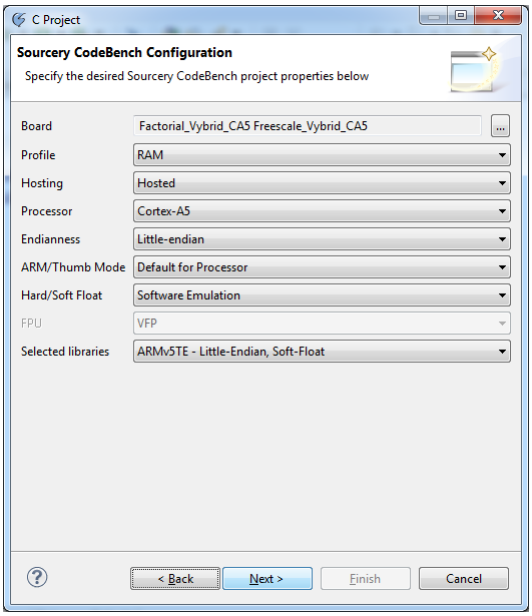


Figure 93: C Project

8. Choose **Debug Interface** as **Ashling Vitra-XD** and choose specific Vitra-XD in **Device**. Click **Finish**. This will create a new project which can be compiled for Freescale Vybrid board Cortex-A5 core.

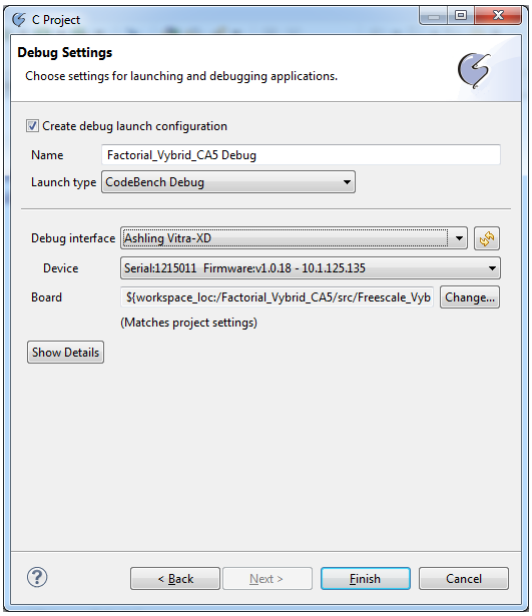


Figure 94: Debug settings

9. To specify the memory map for the Cortex-A5 core select the linker file **Project Explorer|<*Project*>|src|Freescale_Vybrid_CA5_ram.ld** or **Freescale_Vybrid_CA5_ram-hosted.ld**(supports semi-hosting).

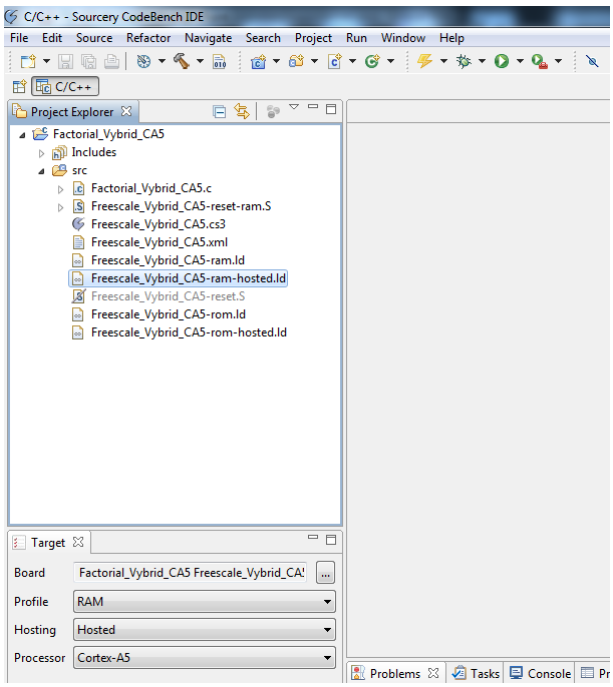


Figure 95: Linker file

10. Configure the memory map appropriately in the selected linker file.

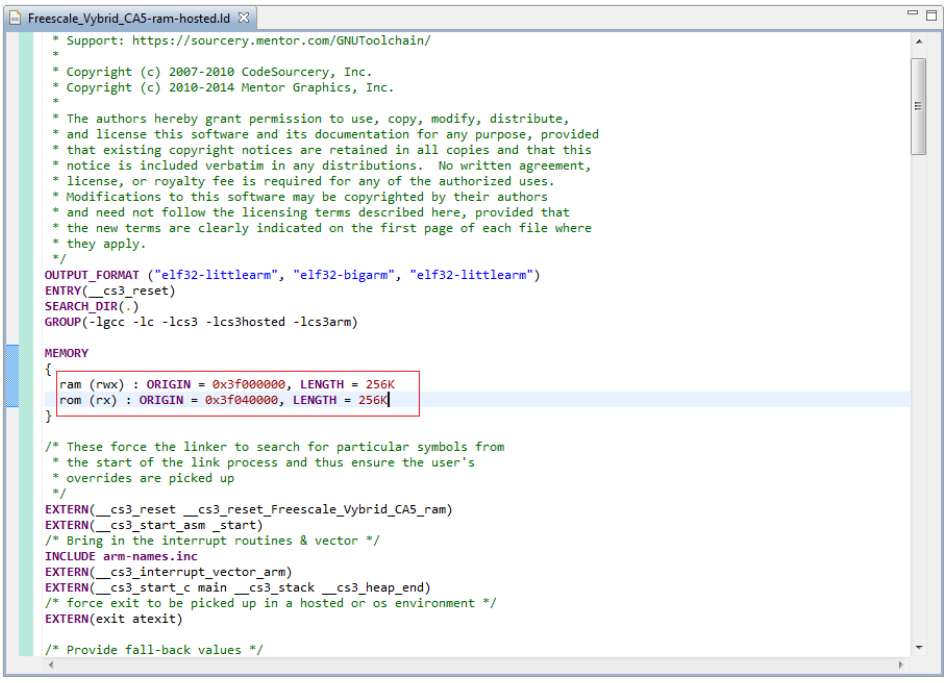
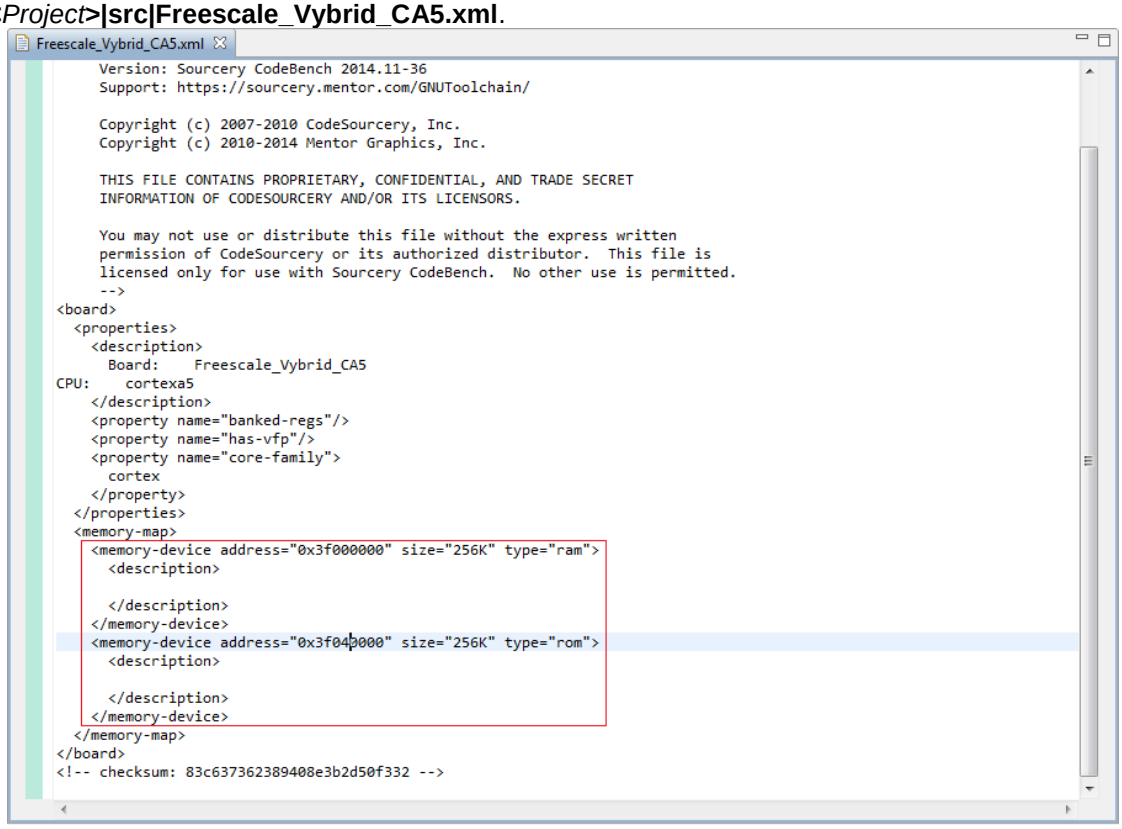


Figure 96: Memory map in linker file

11. Specify the memory map in the created board configuration file **Project Explorer|<*Project*>|src|Freescale_Vybrid_CA5.xml**.




```
<board>
  <properties>
    <description>
      Board:    Freescale_Vybrid_CA5
CPU:    cortexa5
    </description>
    <property name="banked-regs"/>
    <property name="has-vfp"/>
    <property name="core-family">
      cortex
    </property>
  </properties>
  <memory-map>
    <memory-device address="0x3f000000" size="256K" type="ram">
      <description>

      </description>
    </memory-device>
    <memory-device address="0x3f040000" size="256K" type="rom">
      <description>

      </description>
    </memory-device>
  </memory-map>
</board>
<!-- checksum: 83c637362389408e3b2d50f332 -->
```

Figure 97: Memory map in board configuration file

12. Now the project is ready to be compiled and debugged.

Note: While cleaning the project for the first time, a pop-up will appear. Select **No** and proceed.

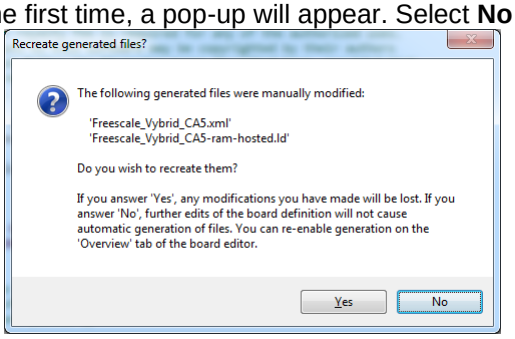


Figure 98: Prompt to clear manual modifications

9. Using Vitra-XD with the TI Sitara AM335x board

9.1 Introduction

This section describes using the TI Sitara AM335x (TMDXEVM3358) board and Vitra-XD to debug and trace software running on the ARM Cortex-A8 core.

9.2 Connecting Vitra-XD to TI Sitara AM335x

The Vitra-XD is connected to the 20-pin CTI connector on the TI Sitara board using the ARM38-CTI20 adapter provided by Ashling as shown below:

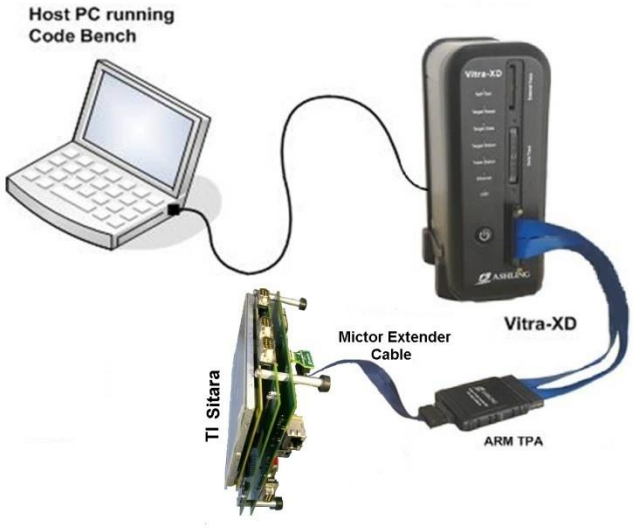


Figure 99: Vitra-XD/Sitara hardware setup

9.3 Software setup

9.3.1 Debugging

Ashling provide a suitable demo project named `SitaraDemo` for the Sitara board in `<Installation_path>\codebench\i686-mingw32\arm-none-eabi\ash\demos>`.This program is built to run from external ARM at 0x80300000 which is an external DDR. This DDR must be initialised before downloading and a GDB script is provided to do this (see below) and uses UART0 to send messages. An STM demo program is also located in `<Installation_path>\codebench\i686-mingw32\arm-none-eabi\ash\demos\stm_demos\am335x>`

To import, build and debug the required project please refer to **section3.2.1**.*Note*: Choose the **Target Adaptor** as **mictor38** and select the **Device** as**TI_AM3359** in the ***Debugger|Ashling Vitra-XD*** as shown below when debugging:

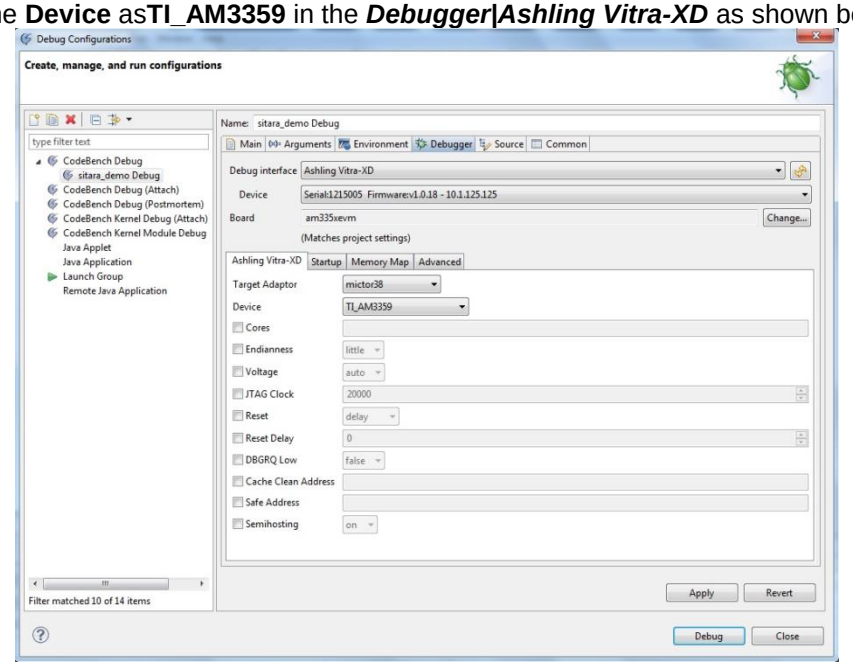


Figure 100: Debug Configuration

As previously mentioned external DDR must be initialized before downloading the `SitaraDemo` program. A suitable GDB script is provided to initialize the TI Sitara AM335x DDR at: `<Installation_path>\codebench\i686-mingw32\arm-none-eabi\ash\scripts\am335x\am335x_ddr_init.gdb>`.

Enter this in **Debug Configurations|Debugger|Startup** using GDB's *source* command as shown below (don't forget to prefix with *source)*:

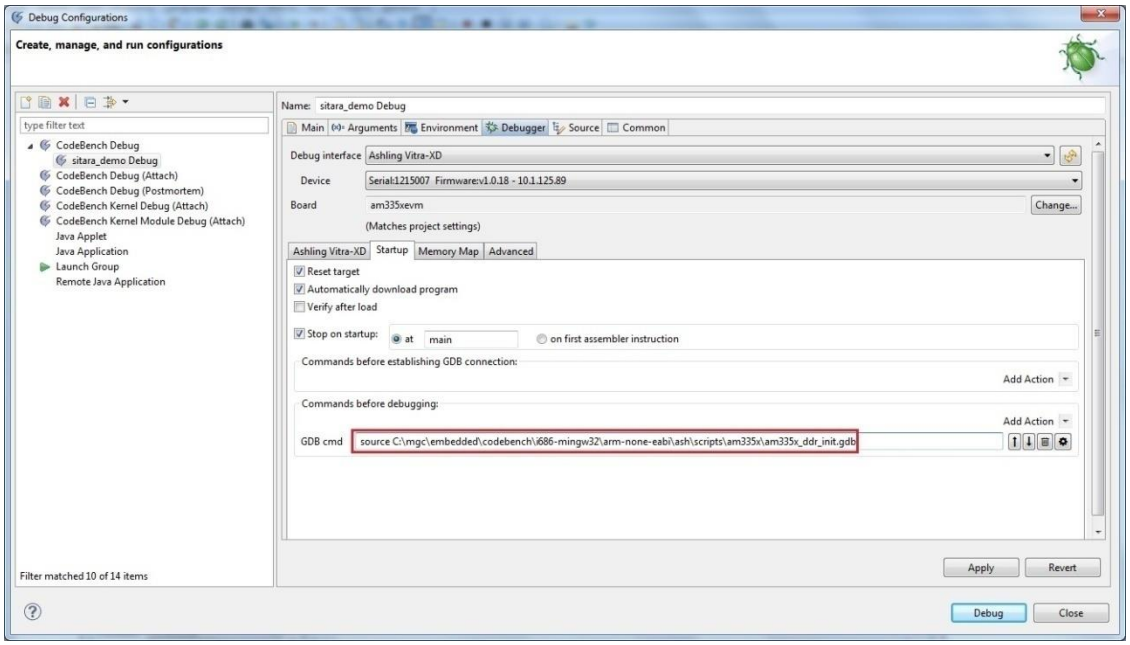


Figure 101: Debug Configuration

9.3.2 Tracing

The TI Sitara AM335x supports both:

- Embedded Trace Buffer(ETB)
  Trace is captured and stored on-chip and uploaded to Vitra-XD via the JTAG interface. Requires a JTAG connection only between the Sitara board and Vitra-XD (e.g. 20-pin CTI (0.05" pitch) or 20-pin IDC (legacy, 0.1" pitch)). Refer to**section4.4** for more details on ETB. When using ETB, ensure you set **Trace port clocking mode** to **Normal** as ETB does not support other modes

- System Trace Macrocell(STM)
  Real-time software instrumentation data is emitted via the AM335x EMU pins and is captured by the Vitra-XD. STM requires both JTAG and EMU pins i.e. 20-pin CTI or 38-way Mictor connector on your target board. Refer to**section4.3** for more details on STM. *Note*: EMU pins are not enabled in STM mode by default on the Sitara board and need to be enabled by running the provided GDB script: `<Installation_path>\codebench\i686-mingw32\arm-none-eabi\ash\gdbscripts\am335x>`. Refer to **section 4.1** for more details.

STM trace can be quickly configured via **Trace|QuickTrace|Trace up to last instruction**.

10. Vitra-XD Firmware upgrade

Sourcery CodeBench allows users to manually upgrade the firmware of the Vitra-XD debug probe. Manual firmware upgrade can be done through Sourcery CodeBench's **Run|AshlingVitra-XD|Firmwareupgrade** menu item (only accessible when you are not connected to a target). If the Vitra-XD firmware version is older than the version in the Sourcery CodeBench directory (on your host PC), then you are prompted to upgrade. Upgrading when prompted is strongly recommended as newer versions of Sourcery CodeBench may not function with older versions of firmware.

11. Ashling Vitra-XD Help menu

This APB can be opened from Sourcery CodeBench's **Help|Ashling Vitra-XD Help** or **Trace|Ashling Vitra-XD Help**.

12. Conclusion

This APB shows the debugging capabilities of Vitra-XD debug/trace probe when used in-conjunction with Sourcery CodeBench. Powerful features such as real-time trace capture are easily configured and used from within Sourcery CodeBench's user-interface. These features allow real-time, non-intrusive debug and analysis of your OMAP4460/OMAP4430 based embedded application, thus helping you to achieve on-time delivery to market. We hope you like it! Please send your feedback to hugh.okeeffe@nestgroup.net

13. Appendix A. Vitra-XD LEDs

The following table describes the LEDs on the Vitra-XD:

LED	State	Meaning
Self Test	Off	No Power connected to Vitra-XD
	Orange Blinking	Power-On-Self-Test (POST) in progress
	Green Blinking	Power-On-Self-Test (POST) OK. Vitra-XD booting in progress
	Green Permanent	Vitra-XD booted OK
	Red	Internal error
Target Reset	Off	Target reset not asserted
	Red Blink	Target reset asserted
Target Data	Off	No traffic between Vitra-XD and Target
	Green	Traffic between Vitra-XD and Target (e.g. TDI).
	Red	Traffic between Target and Vitra-XD (e.g. TDO)
	Orange	Traffic in both directions (normal operation)
Target Status	Off	Target reference voltage not detected.
	Green	Target reference voltage detected
	Red Blink	Target status changed e.g. go/stop or stop/go
Trace Status	Off	Vitra-XD Trace not configured
	Green	Vitra-XD Trace configured (Ready)
	Red	Vitra-XD Trace buffer full
	Orange	Trace in progress
Ethernet	Off	Ethernet not selected
	Green	Ethernet selected
	Red	Ethernet error
	Orange Blink	Ethernet transactions ongoing
USB	Off	USB not selected
	Green	USB selected
	Red	USB error
	Orange Blink	USB transactions ongoing

Table 2. Vitra-XD LEDs

14. Appendix B. CE Notice

The **CE** mark on the back of this Ashling product indicates its compliance with the EMC (Electromagnetic Compatibility) Directive of the European Union (Directive 2004/108/EC). In accordance with this directive, this Ashling product has been tested to the following technical standards:

- EN 61326-1:2006: Electrical equipment for measurement, control and laboratory use.
- Equipment classification: Class B (domestic and light industrial)

To ensure the continued compliance of your Ashling product with the EMC directive (and to ensure that your product can be used without causing interference to, or being affected by other electronic equipment), please note the following:

- This Ashling product is intended for use in the development and test of electronic systems in a development laboratory, by suitably trained staff.
- This Ashling product has been designed to be used with a target system. It should be noted that there may be exposed electronic circuitry on the target system, thus when handling the target please note that it is possible that electrostatic discharges (ESD) can potentially cause damage to the target or, due to the cabling connection, to the Vitra-XD itself. Please exercise all the normal precautions required for electrostatic sensitive devices when handling the target system including the use of a workbench equipped to control static electricity and an anti-static wrist strap, properly connected to the workbench.
- This product is designed for use with a Personal Computer or Laptop whose electromagnetic emission and susceptibility performance comply with the EMC Directive.
- This product is designed for use with an external 12V DC supply whose electromagnetic emission and susceptibility performance comply with the EMC Directive.

15. Appendix C. RAW Trace Format

RAW trace information includes the Source-ID, Port data and Timestamp in a binary format as follows:

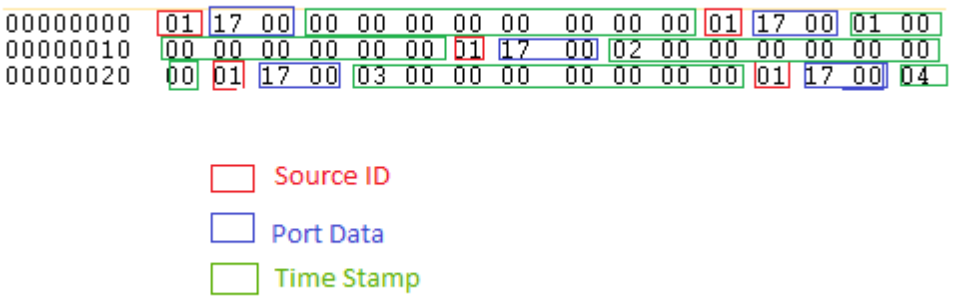


Figure 102: RAW Trace Format

I.e. one byte Source ID, two bytes port data followed by eight bytes timestamp.

15.1 Source ID

Source ID indicates the trace source (ARM core) from which the trace data was generated. For example, *Source ID* 1 refers to the trace data generated by core 0 and a value of 2 refers to trace data generated by core 1. For more details, refer to:
http://infocenter.arm.com/help/topic/com.arm.doc.ihi0029d/IHI0029D_coresight_architecture_spec_v2_0.pdf.

15.2 Port Data

Port Data refers to the raw trace information obtained from ARM core. Port Data is dependent on the ARM trace protocol supported by the target. For ETM refer to:
http://infocenter.arm.com/help/topic/com.arm.doc.ihi0014q/IHI0014Q_etm_architecture_spec.pdf

and for PTM:

http://infocenter.arm.com/help/topic/com.arm.doc.ihi0035b/IHI0035B_cs_pft_v1_1_architecture_spec.pdf

15.3 Timestamp

A 64-bit timestamp in units of 5 nanoseconds e.g. a value of 1 indicates 5ns

15.4 Sample 'C' code to access Timestamp

```
typedef struct
{
   /*! Frame id */
   unsigned long long ullFrameId;
   /*! Source id - The source from where the trace packet was generated. */
   unsigned int uiSrcID;
   /*! Port data */
   unsigned int uiPortData;
   /*! Timestamp */
   unsigned long long ullTimeStamp;
   /*! Indicate whether timestamp is updated or not. */
   unsigned int uiTimeStampUpdated;
}TyBusTrace;

int main()
{
        TyBusTrace *ptyBusTracePtm=NULL;
        FILE* pFilePtrbin = NULL;
        static char szBusTraceFileNameBinary[256];
        int iIndex=0;

        /* Allocating memory for the structure*/
        /*Let's assume that the number of frames captured is 500 */
        ptyBusTracePtm =(TyBusTrace *)malloc(500*sizeof(TyBusTrace));

        /*Assuming the file is in the static path E:\\Savetracebintest */
        FOPEN(pFilePtrbin, "E:\\Savetracebintest","rb");
        if(NULL == pFilePtrbin)
```

```
        {
                perror("Fopen failed :");
                return -1;
        }

        for(iIndex = 0;iIndex<500 ;iIndex++)
        {
                ptyBusTracePtm[iIndex].ullFrameId = iIndex;
                  fread(&ptyBusTracePtm[iIndex].uiSrcID , sizeof(unsigned char) ,1,
        pFilePtrbin );
                  fread(&ptyBusTracePtm[iIndex].uiPortData , sizeof(unsigned short) , 1                ,
pFilePtrbin );
                  fread(&ptyBusTracePtm[iIndex].ullTimeStamp ,
                sizeof(ptyBusTracePtm[iIndex].ullTimeStamp) , 1 ,                       pFilePtrbin );
        }

        fclose(pFilePtrbin);
        free(ptyBusTracePtm);
        return 0;
}
```

Doc: APB217-Vitra-XD(with CB).docx